PEARSON

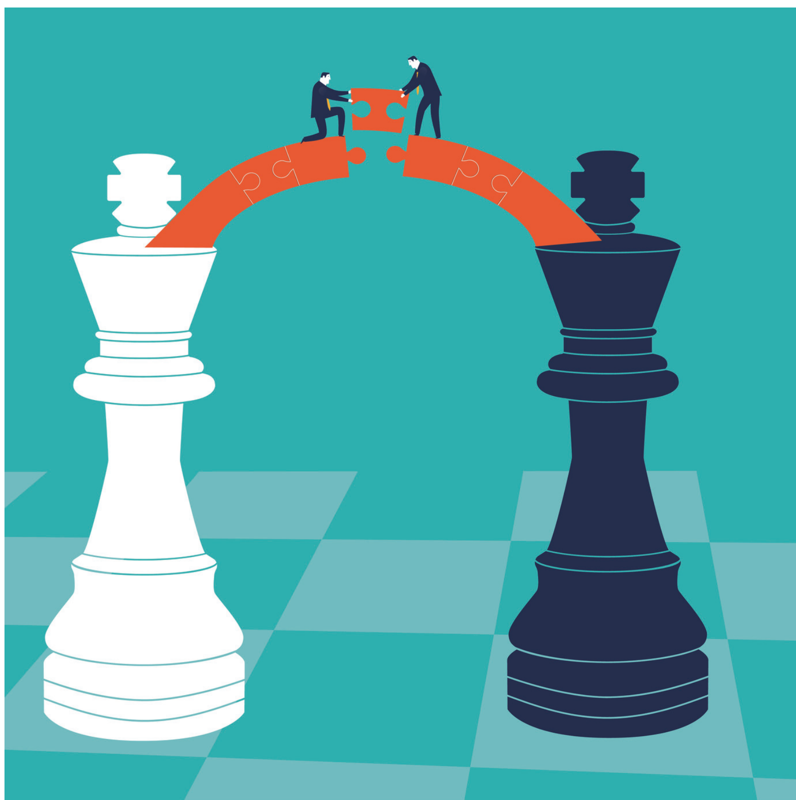
C和C++实务精选
品味岁月积淀，读享技术菁华

C Primer Plus

（第6版）中文版

C Primer Plus
Sixth Edition

- 经久不衰的C语言畅销经典教程
- 针对C11标准进行全面更新



[美] | Stephen Prata | 著
姜佑 | 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

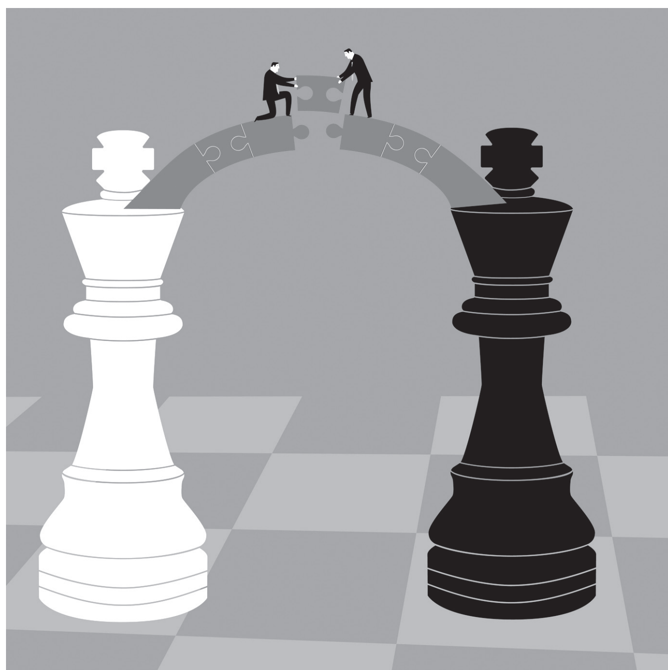
PEARSON

C Primer Plus

（第6版）中文版

[美] | Stephen Prata | 著
姜佑 | 译

C Primer Plus
Sixth Edition



人 民 邮 电 出 版 社
北 京

异步社区电子书

感谢您购买异步社区电子书！异步社区已上架电子书 500 余种，社区还会经常发布福利信息，对社区有贡献的读者赠送免费样书券、优惠码、积分等等，希望您在阅读过程中，把您的阅读体验传递给我们，让我们了解读者心声，有问题我们会及时修正。

社区网址： http://www.epubit.com.cn/

反馈邮箱： contact@epubit.com.cn

异步社区里有什么？

图书、电子书（半价电子书）、优秀作译者、访谈、技术会议播报、赠书活动、下载资源。

异步社区特色：

纸书、电子书同步上架、纸电捆绑超值优惠购买。

最新精品技术图书全网首发预售。

晒单有意外惊喜！

异步社区里可以做什么？

博客式写作发表文章，提交勘误赚取积分，积分兑换样书，写书评赢样书券等。

联系我们：

微博：

@ 人邮异步社区　　@ 人民邮电出版社 - 信息技术分社

微信公众号：

人邮 IT 书坊　　异步社区

QQ 群： 368449889

图书在版编目（ＣＩＰ）数据

C Primer Plus（第6版）中文版 / (美) 普拉达(Prata,S.) 著 ; 姜佑译. -- 北京 : 人民邮电出版社, 2016.4
ISBN 978-7-115-39059-2

Ⅰ. ①C… Ⅱ. ①普… ②姜… Ⅲ. ①C语言—程序设计 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第084602号

版权声明

◆ 著　　　　［美］Stephen Prata
　译　　　　姜　佑
　责任编辑　傅道坤
　责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路 11 号
　邮编　100164　　电子邮件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京圣夫亚美印刷有限公司印刷

◆ 开本：787×1092　1/16
　印张：47
　字数：1412 千字　　　　2016 年 4 月第 1 版
　印数：1 – 8 000 册　　　2016 年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2014-5617 号

定价：89.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316
反盗版热线：(010)81055315

内容提要

本书详细讲解了 C 语言的基本概念和编程技巧。

全书共 17 章。第 1 章、第 2 章介绍了 C 语言编程的预备知识。第 3 章～第 15 章详细讲解了 C 语言的相关知识，包括数据类型、格式化输入/输出、运算符、表达式、语句、循环、字符输入和输出、函数、数组和指针、字符和字符串函数、内存管理、文件输入输出、结构、位操作等。第 16 章、第 17 章介绍 C 预处理器、C 库和高级数据表示。本书以完整的程序为例，讲解 C 语言的知识要点和注意事项。每章末尾设计了大量复习题和编程练习，帮助读者巩固所学知识和提高实际编程能力。附录给出了各章复习题的参考答案和丰富的参考资料。

本书可作为 C 语言的教材，适用于需要系统学习 C 语言的初学者，也适用于巩固 C 语言知识或希望进一步提高编程技术的程序员。

作者简介

Stephen Prata 曾在加利福尼亚的马林学院（肯特菲尔德）教授天文学、物理学和程序设计课程，现已退休。他在加州理工学院获得学士学位，在加州大学伯克利分校获得博士学位。他最早接触程序设计，是为了利用计算机给星团建模。Stephen 撰写和与他人合著了十几本图书，其中包括 *C++ Primer Plus* 和 *UNIX Primer Plus*。

献　辞

谨将本书献给我的父亲 William Prata。

致谢

感谢 Pearson 的 Mark Taber 一直都非常关注本书。感谢 Danny Kalev 在技术上提供的帮助和建议。

前言

1984 年 *C Primer Plus* 第 1 版刚问世时，使用 C 语言编程的人并不多。C 语言从那时开始流行，许多人在本书的帮助下掌握了 C 语言。实际上，*C Primer Plus* 各个版本累计销售量已超过 55 万册。

C 语言从早期的非正式的 K&R 标准，发展到 1990 ISO/ANSI 标准，进而发展到 2011 ISO/IEC 标准。本书也随着逐渐成熟，发展到现在的第 6 版。在所有这些版本中，我的目标是致力于编写一本指导性强、条理清晰而且有用的 C 语言教程。

本书的用法和目标

我希望撰写一本友好、方便使用、便于自学的指南。为此，本书采用以下写作策略。

- 在介绍 C 语言细节的同时，讲解编程概念。本书假定读者为非专业的程序员。
- 每次尽量用短小简单的示例演示一两个概念，学以致用是最有效的学习方式之一。
- 当概念用文字较难解释时，则用图表演示以帮助读者理解。
- C 语言的主要特性总结在方框中，便于查找和复习。
- 每章末尾设有复习题和编程练习，帮助读者测试和加深对 C 语言的理解。

为了获得最佳的学习效果，学习本书时，读者应尽量扮演一个积极的角色。不仅要仔细阅读程序示例，还要亲自动手录入程序并运行。C 是一种可移植性很高的语言，但有时在你的系统中运行的结果和在我们的系统中运行的结果不同。经常改动程序的某些部分，运行后看看有什么效果。偶尔出现警告也不必理会，主要是看一下执行错误操作会出现什么状况。在学习的过程中应该多提出问题和多练习。用得越多，学的知识就越牢固。

希望本书能帮助读者轻松愉快地学习 C 语言。

目录

第 1 章　初识 C 语言······1

1.1　C 语言的起源······1

1.2　选择 C 语言的理由······1

1.2.1　设计特性······1

1.2.2　高效性······1

1.2.3　可移植性······2

1.2.4　强大而灵活······3

1.2.5　面向程序员······3

1.2.6　缺点······3

1.3　C 语言的应用范围······3

1.4　计算机能做什么······4

1.5　高级计算机语言和编译器······5

1.6　语言标准······6

1.6.1　第 1 个 ANSI/ISO C 标准······6

1.6.2　C99 标准······6

1.6.3　C11 标准······7

1.7　使用 C 语言的 7 个步骤······7

1.7.1　第 1 步：定义程序的目标······8

1.7.2　第 2 步：设计程序······8

1.7.3　第 3 步：编写代码······8

1.7.4　第 4 步：编译······8

1.7.5　第 5 步：运行程序······9

1.7.6　第 6 步：测试和调试程序······9

1.7.7　第 7 步：维护和修改代码······9

1.7.8　说明······9

1.8　编程机制······10

1.8.1　目标代码文件、可执行文件和库······10

1.8.2　UNIX 系统······11

1.8.3　GNU 编译器集合和 LLVM 项目······13

1.8.4 Linux 系统 ······ 13
1.8.5 PC 的命令行编译器 ······ 14
1.8.6 集成开发环境（Windows） ······ 14
1.8.7 Windows/Linux ······ 15
1.8.8 Macintosh 中的 C ······ 15
1.9 本书的组织结构 ······ 15
1.10 本书的约定 ······ 16
1.10.1 字体 ······ 16
1.10.2 程序输出 ······ 16
1.10.3 特殊元素 ······ 17
1.11 本章小结 ······ 17
1.12 复习题 ······ 18
1.13 编程练习 ······ 18

第 2 章 C 语言概述 ······ 19

2.1 简单的 C 程序示例 ······ 19
2.2 示例解释 ······ 20
2.2.1 第 1 遍：快速概要 ······ 21
2.2.2 第 2 遍：程序细节 ······ 21
2.3 简单程序的结构 ······ 28
2.4 提高程序可读性的技巧 ······ 28
2.5 进一步使用 C ······ 29
2.5.1 程序说明 ······ 30
2.5.2 多条声明 ······ 30
2.5.3 乘法 ······ 30
2.5.4 打印多个值 ······ 30
2.6 多个函数 ······ 30
2.7 调试程序 ······ 32
2.7.1 语法错误 ······ 32
2.7.2 语义错误 ······ 33
2.7.3 程序状态 ······ 34
2.8 关键字和保留标识符 ······ 34
2.9 关键概念 ······ 35
2.10 本章小结 ······ 35
2.11 复习题 ······ 36

2.12 编程练习······37

第 3 章 数据和 C······39

3.1 示例程序······39
3.2 变量与常量数据······42
3.3 数据：数据类型关键字······42
3.3.1 整数和浮点数······43
3.3.2 整数······43
3.3.3 浮点数······43
3.4 C 语言基本数据类型······44
3.4.1 int 类型······44
3.4.2 其他整数类型······47
3.4.3 使用字符：char 类型······50
3.4.4 _Bool 类型······54
3.4.5 可移植类型：stdint.h 和 inttypes.h······55
3.4.6 float、double 和 long double······56
3.4.7 复数和虚数类型······60
3.4.8 其他类型······60
3.4.9 类型大小······62
3.5 使用数据类型······63
3.6 参数和陷阱······63
3.7 转义序列示例······64
3.7.1 程序运行情况······65
3.7.2 刷新输出······65
3.8 关键概念······66
3.9 本章小结······66
3.10 复习题······67
3.11 编程练习······68

第 4 章 字符串和格式化输入/输出······71

4.1 前导程序······71
4.2 字符串简介······72
4.2.1 char 类型数组和 null 字符······72
4.2.2 使用字符串······73
4.2.3 strlen()函数······74

4.3 常量和 C 预处理器……76
4.3.1 const 限定符……78
4.3.2 明示常量……78
4.4 printf()和 scanf()……80
4.4.1 printf()函数……80
4.4.2 使用 printf()……81
4.4.3 printf()的转换说明修饰符……83
4.4.4 转换说明的意义……87
4.4.5 使用 scanf()……92
4.4.6 printf()和 scanf()的*修饰符……95
4.4.7 printf()的用法提示……97
4.5 关键概念……98
4.6 本章小结……98
4.7 复习题……99
4.8 编程练习……100
第 5 章 运算符、表达式和语句……103
5.1 循环简介……103
5.2 基本运算符……105
5.2.1 赋值运算符：=……105
5.2.2 加法运算符：+……107
5.2.3 减法运算符：-……107
5.2.4 符号运算符：-和+……107
5.2.5 乘法运算符：*……108
5.2.6 除法运算符：/……110
5.2.7 运算符优先级……110
5.2.8 优先级和求值顺序……112
5.3 其他运算符……113
5.3.1 sizeof 运算符和 size_t 类型……113
5.3.2 求模运算符：%……114
5.3.3 递增运算符：++……115
5.3.4 递减运算符：--……118
5.3.5 优先级……118
5.3.6 不要自作聪明……119
5.4 表达式和语句……120

5.4.1 表达式……120
5.4.2 语句……120
5.4.3 复合语句（块）……123
5.5 类型转换……124
5.6 带参数的函数……127
5.7 示例程序……129
5.8 关键概念……130
5.9 本章小结……130
5.10 复习题……131
5.11 编程练习……134

第 6 章 C 控制语句：循环……137

6.1 再探 while 循环……137
6.1.1 程序注释……138
6.1.2 C 风格读取循环……139
6.2 while 语句……140
6.2.1 终止 while 循环……140
6.2.2 何时终止循环……141
6.2.3 while：入口条件循环……141
6.2.4 语法要点……141
6.3 用关系运算符和表达式比较大小……143
6.3.1 什么是真……144
6.3.2 其他真值……145
6.3.3 真值的问题……146
6.3.4 新的_Bool 类型……147
6.3.5 优先级和关系运算符……148
6.4 不确定循环和计数循环……150
6.5 for 循环……151
6.6 其他赋值运算符：+=、-=、*=、/=、%=……155
6.7 逗号运算符……156
6.8 出口条件循环：do while……159
6.9 如何选择循环……161
6.10 嵌套循环……162
6.10.1 程序分析……163
6.10.2 嵌套变式……163

6.11 数组简介……164
6.12 使用函数返回值的循环示例……166
6.12.1 程序分析……168
6.12.2 使用带返回值的函数……169
6.13 关键概念……169
6.14 本章小结……170
6.15 复习题……170
6.16 编程练习……174

第 7 章 C 控制语句：分支和跳转……177

7.1 if 语句……177
7.2 if else 语句……179
7.2.1 另一个示例：介绍 getchar()和 putchar()……180
7.2.2 ctype.h 系列的字符函数……182
7.2.3 多重选择 else if……184
7.2.4 else 与 if 配对……186
7.2.5 多层嵌套的 if 语句……187
7.3 逻辑运算符……190
7.3.1 备选拼写：iso646.h 头文件……191
7.3.2 优先级……192
7.3.3 求值顺序……192
7.3.4 范围……193
7.4 一个统计单词的程序……194
7.5 条件运算符：?:……196
7.6 循环辅助：continue 和 break……198
7.6.1 continue 语句……198
7.6.2 break 语句……200
7.7 多重选择：switch 和 break……202
7.7.1 switch 语句……204
7.7.2 只读每行的首字符……205
7.7.3 多重标签……206
7.7.4 switch 和 if else……208
7.8 goto 语句……208
7.9 关键概念……211
7.10 本章小结……211

7.11 复习题……212
7.12 编程练习……214

第 8 章 字符输入/输出和输入验证……217

8.1 单字符 I/O：getchar()和 putchar()……217
8.2 缓冲区……218
8.3 结束键盘输入……219
8.3.1 文件、流和键盘输入……219
8.3.2 文件结尾……220
8.4 重定向和文件……222
8.5 创建更友好的用户界面……226
8.5.1 使用缓冲输入……226
8.5.2 混合数值和字符输入……228
8.6 输入验证……230
8.6.1 分析程序……234
8.6.2 输入流和数字……234
8.7 菜单浏览……235
8.7.1 任务……235
8.7.2 使执行更顺利……235
8.7.3 混合字符和数值输入……237
8.8 关键概念……240
8.9 本章小结……240
8.10 复习题……241
8.11 编程练习……241

第 9 章 函数……243

9.1 复习函数……243
9.1.1 创建并使用简单函数……244
9.1.2 分析程序……245
9.1.3 函数参数……247
9.1.4 定义带形式参数的函数……248
9.1.5 声明带形式参数函数的原型……249
9.1.6 调用带实际参数的函数……249
9.1.7 黑盒视角……250
9.1.8 使用 return 从函数中返回值……250

9.1.9 函数类型……252
9.2 ANSI C 函数原型……253
9.2.1 问题所在……253
9.2.2 ANSI 的解决方案……254
9.2.3 无参数和未指定参数……255
9.2.4 函数原型的优点……256
9.3 递归……256
9.3.1 演示递归……256
9.3.2 递归的基本原理……258
9.3.3 尾递归……258
9.3.4 递归和倒序计算……260
9.3.5 递归的优缺点……262
9.4 编译多源代码文件的程序……262
9.4.1 UNIX……263
9.4.2 Linux……263
9.4.3 DOS 命令行编译器……263
9.4.4 Windows 和苹果的 IDE 编译器……263
9.4.5 使用头文件……263
9.5 查找地址：&运算符……267
9.6 更改主调函数中的变量……268
9.7 指针简介……269
9.7.1 间接运算符：*……270
9.7.2 声明指针……270
9.7.3 使用指针在函数间通信……271
9.8 关键概念……274
9.9 本章小结……275
9.10 复习题……275
9.11 编程练习……276

第 10 章 数组和指针……277

10.1 数组……277
10.1.1 初始化数组……277
10.1.2 指定初始化器（C99）……281
10.1.3 给数组元素赋值……282
10.1.4 数组边界……282

10.1.5 指定数组的大小……284
10.2 多维数组……284
10.2.1 初始化二维数组……287
10.2.2 其他多维数组……288
10.3 指针和数组……288
10.4 函数、数组和指针……290
10.4.1 使用指针形参……293
10.4.2 指针表示法和数组表示法……294
10.5 指针操作……295
10.6 保护数组中的数据……298
10.6.1 对形式参数使用 const……299
10.6.2 const 的其他内容……300
10.7 指针和多维数组……302
10.7.1 指向多维数组的指针……304
10.7.2 指针的兼容性……305
10.7.3 函数和多维数组……306
10.8 变长数组（VLA）……309
10.9 复合字面量……312
10.10 关键概念……314
10.11 本章小结……315
10.12 复习题……316
10.13 编程练习……317
第 11 章 字符串和字符串函数……321
11.1 表示字符串和字符串 I/O……321
11.1.1 在程序中定义字符串……322
11.1.2 指针和字符串……328
11.2 字符串输入……329
11.2.1 分配空间……329
11.2.2 不幸的 gets()函数……330
11.2.3 gets()的替代品……331
11.2.4 scanf()函数……336
11.3 字符串输出……337
11.3.1 puts()函数……338
11.3.2 fputs()函数……339

11.3.3 printf()函数……339
11.4 自定义输入/输出函数……340
11.5 字符串函数……342
11.5.1 strlen()函数……342
11.5.2 strcat()函数……343
11.5.3 strncat()函数……345
11.5.4 strcmp()函数……346
11.5.5 strcpy()和 strncpy()函数……351
11.5.6 sprintf()函数……356
11.5.7 其他字符串函数……357
11.6 字符串示例：字符串排序……359
11.6.1 排序指针而非字符串……360
11.6.2 选择排序算法……361
11.7 ctype.h 字符函数和字符串……362
11.8 命令行参数……363
11.8.1 集成环境中的命令行参数……365
11.8.2 Macintosh 中的命令行参数……365
11.9 把字符串转换为数字……365
11.10 关键概念……368
11.11 本章小结……368
11.12 复习题……369
11.13 编程练习……371

第 12 章 存储类别、链接和内存管理……373

12.1 存储类别……373
12.1.1 作用域……374
12.1.2 链接……376
12.1.3 存储期……376
12.1.4 自动变量……377
12.1.5 寄存器变量……380
12.1.6 块作用域的静态变量……381
12.1.7 外部链接的静态变量……382
12.1.8 内部链接的静态变量……386
12.1.9 多文件……386
12.1.10 存储类别说明符……387

12.1.11 存储类别和函数 389
12.1.12 存储类别的选择 389
12.2 随机数函数和静态变量 390
12.3 掷骰子 393
12.4 分配内存：malloc()和 free() 396
12.4.1 free()的重要性 399
12.4.2 calloc()函数 400
12.4.3 动态内存分配和变长数组 400
12.4.4 存储类别和动态内存分配 401
12.5 ANSI C 类型限定符 402
12.5.1 const 类型限定符 403
12.5.2 volatile 类型限定符 404
12.5.3 restrict 类型限定符 405
12.5.4 _Atomic 类型限定符（C11） 406
12.5.5 旧关键字的新位置 406
12.6 关键概念 407
12.7 本章小结 407
12.8 复习题 408
12.9 编程练习 409

第 13 章 文件输入/输出 413

13.1 与文件进行通信 413
13.1.1 文件是什么 413
13.1.2 文本模式和二进制模式 413
13.1.3 I/O 的级别 415
13.1.4 标准文件 415
13.2 标准 I/O 415
13.2.1 检查命令行参数 416
13.2.2 fopen()函数 416
13.2.3 getc()和 putc()函数 417
13.2.4 文件结尾 418
13.2.5 fclose()函数 419
13.2.6 指向标准文件的指针 419
13.3 一个简单的文件压缩程序 419
13.4 文件 I/O：fprintf()、fscanf()、fgets()和 fputs() 421

13.4.1 fprintf()和 fscanf()函数 ……421
13.4.2 fgets()和 fputs()函数……422
13.5 随机访问：fseek()和 ftell() ……423
13.5.1 fseek()和 ftell()的工作原理……424
13.5.2 二进制模式和文本模式 ……425
13.5.3 可移植性……425
13.5.4 fgetpos()和 fsetpos()函数 ……426
13.6 标准 I/O 的机理 ……426
13.7 其他标准 I/O 函数 ……427
13.7.1 int ungetc(int c, FILE *fp)函数……427
13.7.2 int fflush()函数 ……428
13.7.3 int setvbuf()函数……428
13.7.4 二进制 I/O：fread()和 fwrite() ……428
13.7.5 size_t fwrite()函数 ……429
13.7.6 size_t fread()函数 ……430
13.7.7 int feof(FILE *fp)和 int ferror(FILE *fp)函数……430
13.7.8 一个程序示例……430
13.7.9 用二进制 I/O 进行随机访问……433
13.8 关键概念……435
13.9 本章小结……435
13.10 复习题……435
13.11 编程练习……437

第 14 章 结构和其他数据形式……439

14.1 示例问题：创建图书目录……439
14.2 建立结构声明……441
14.3 定义结构变量……441
14.3.1 初始化结构……442
14.3.2 访问结构成员 ……443
14.3.3 结构的初始化器 ……443
14.4 结构数组……444
14.4.1 声明结构数组……446
14.4.2 标识结构数组的成员 ……447
14.4.3 程序讨论……447
14.5 嵌套结构……448

14.6 指向结构的指针……449
14.6.1 声明和初始化结构指针……450
14.6.2 用指针访问成员……451
14.7 向函数传递结构的信息……451
14.7.1 传递结构成员……451
14.7.2 传递结构的地址……452
14.7.3 传递结构……453
14.7.4 其他结构特性……454
14.7.5 结构和结构指针的选择……458
14.7.6 结构中的字符数组和字符指针……458
14.7.7 结构、指针和 malloc()……459
14.7.8 复合字面量和结构（C99）……462
14.7.9 伸缩型数组成员（C99）……463
14.7.10 匿名结构（C11）……465
14.7.11 使用结构数组的函数……466
14.8 把结构内容保存到文件中……467
14.8.1 保存结构的程序示例……468
14.8.2 程序要点……470
14.9 链式结构……471
14.10 联合简介……472
14.10.1 使用联合……472
14.10.2 匿名联合（C11）……473
14.11 枚举类型……474
14.11.1 enum 常量……475
14.11.2 默认值……475
14.11.3 赋值……475
14.11.4 enum 的用法……476
14.11.5 共享名称空间……477
14.12 typedef 简介……478
14.13 其他复杂的声明……479
14.14 函数和指针……481
14.15 关键概念……487
14.16 本章小结……487
14.17 复习题……488
14.18 编程练习……490

第 15 章　位操作……493

15.1　二进制数、位和字节……493

15.1.1　二进制整数……494

15.1.2　有符号整数……494

15.1.3　二进制浮点数……495

15.2　其他进制数……495

15.2.1　八进制……495

15.2.2　十六进制……496

15.3　C 按位运算符……496

15.3.1　按位逻辑运算符……497

15.3.2　用法：掩码……498

15.3.3　用法：打开位（设置位）……498

15.3.4　用法：关闭位（清空位）……499

15.3.5　用法：切换位……499

15.3.6　用法：检查位的值……500

15.3.7　移位运算符……500

15.3.8　编程示例……501

15.3.9　另一个例子……503

15.4　位字段……505

15.4.1　位字段示例……506

15.4.2　位字段和按位运算符……509

15.5　对齐特性（C11）……515

15.6　关键概念……516

15.7　本章小结……516

15.8　复习题……517

15.9　编程练习……518

第 16 章　C 预处理器和 C 库……521

16.1　翻译程序的第一步……521

16.2　明示常量：#define……522

16.2.1　记号……525

16.2.2　重定义常量……525

16.3　在#define 中使用参数……525

16.3.1　用宏参数创建字符串：#运算符……527

16.3.2 预处理器黏合剂：##运算符 528

16.3.3 变参宏：...和__VA_ARGS__ 529

16.4 宏和函数的选择 530

16.5 文件包含：#include 531

16.5.1 头文件示例 531

16.5.2 使用头文件 533

16.6 其他指令 534

16.6.1 #undef 指令 534

16.6.2 从 C 预处理器角度看已定义 534

16.6.3 条件编译 535

16.6.4 预定义宏 539

16.6.5 #line 和#error 540

16.6.6 #pragma 540

16.6.7 泛型选择（C11） 541

16.7 内联函数（C99） 542

16.8 _Noreturn 函数（C11） 544

16.9 C 库 544

16.9.1 访问 C 库 544

16.9.2 使用库描述 545

16.10 数学库 546

16.10.1 三角问题 547

16.10.2 类型变体 548

16.10.3 tgmath.h 库（C99） 550

16.11 通用工具库 550

16.11.1 exit()和 atexit()函数 550

16.11.2 qsort()函数 552

16.12 断言库 556

16.12.1 assert 的用法 556

16.12.2 _Static_assert（C11） 557

16.13 string.h 库中的 memcpy()和 memmove() 558

16.14 可变参数：stdarg.h 560

16.15 关键概念 562

16.16 本章小结 562

16.17 复习题 562

16.18 编程练习 563

第 17 章　高级数据表示……567

17.1　研究数据表示……567
17.2　从数组到链表……570
17.2.1　使用链表……572
17.2.2　反思……576
17.3　抽象数据类型（ADT）……576
17.3.1　建立抽象……577
17.3.2　建立接口……578
17.3.3　使用接口……581
17.3.4　实现接口……583
17.4　队列 ADT……589
17.4.1　定义队列抽象数据类型……590
17.4.2　定义一个接口……590
17.4.3　实现接口数据表示……591
17.4.4　测试队列……598
17.5　用队列进行模拟……600
17.6　链表和数组……605
17.7　二叉查找树……608
17.7.1　二叉树 ADT……608
17.7.2　二叉查找树接口……609
17.7.3　二叉树的实现……611
17.7.4　使用二叉树……624
17.7.5　树的思想……628
17.8　其他说明……629
17.9　关键概念……630
17.10　本章小结……630
17.11　复习题……630
17.12　编程练习……631

附录 A　复习题答案……633

附录 B　参考资料……665

B.1　参考资料 I：补充阅读……665
B.2　参考资料 II：C 运算符……667

B.3　参考资料 III：基本类型和存储类别……671
B.4　参考资料 IV：表达式、语句和程序流……675
B.5　参考资料 V：新增 C99 和 C11 的 ANSI C 库……679
B.6　参考资料 VI：扩展的整数类型……714
B.7　参考资料 VII：扩展字符支持……716
B.8　参考资料 VIII：C99/C11 数值计算增强……720
B.9　参考资料 IX：C 和 C++的区别……726

第 1 章
初识 C 语言

本章介绍以下内容：

- C 的历史和特性
- 编写程序的步骤
- 编译器和链接器的一些知识
- C 标准

欢迎来到 C 语言的世界。C 是一门功能强大的专业化编程语言，深受业余编程爱好者和专业程序员的喜爱。本章为读者学习这一强大而流行的语言打好基础，并介绍几种开发 C 程序最可能使用的环境。

我们先来了解 C 语言的起源和一些特性，包括它的优缺点。然后，介绍编程的起源并探讨一些编程的基本原则。最后，讨论如何在一些常见系统中运行 C 程序。

1.1 C 语言的起源

1972 年，贝尔实验室的丹尼斯 • 里奇（*Dennis Ritch*）和肯 • 汤普逊（*Ken Thompson*）在开发 UNIX 操作系统时设计了 C 语言。然而，C 语言不完全是里奇突发奇想而来，他是在 B 语言（汤普逊发明）的基础上进行设计。至于 B 语言的起源，那是另一个故事。C 语言设计的初衷是将其作为程序员使用的一种编程工具，因此，其主要目标是成为有用的语言。

虽然绝大多数语言都以实用为目标，但是通常也会考虑其他方面。例如，Pascal 的主要目标是为更好地学习编程原理提供扎实的基础；而 BASIC 的主要目标是开发出类似英文的语言，让不熟悉计算机的学生轻松学习编程。这些目标固然很重要，但是随着计算机的迅猛发展，它们已经不是主流语言。然而，最初为程序员设计开发的 C 语言，现在已成为首选的编程语言之一。

1.2 选择 C 语言的理由

在过去 40 多年里，C 语言已成为最重要、最流行的编程语言之一。它的成长归功于使用过的人都对它很满意。过去 20 多年里，虽然许多人都从 C 语言转而使用其他编程语言（如，C++、Objective C、Java 等），但是 C 语言仍凭借自身实力在众多语言中脱颖而出。在学习 C 语言的过程中，会发现它的许多优点（见图 1.1）。下面，我们来看看其中较为突出的几点。

1.2.1 设计特性

C 是一门流行的语言，融合了计算机科学理论和实践的控制特性。C 语言的设计理念让用户能轻松地完成自顶向下的规划、结构化编程和模块化设计。因此，用 C 语言编写的程序更易懂、更可靠。

1.2.2 高效性

C 是高效的语言。在设计上，它充分利用了当前计算机的优势，因此 C 程序相对更紧凑，而且运行速

度很快。实际上，C 语言具有通常是汇编语言才具有的微调控制能力（汇编语言是为特殊的中央处理单元设计的一系列内部指令，使用助记符来表示；不同的 CPU 系列使用不同的汇编语言），可以根据具体情况微调程序以获得最大运行速度或最有效地使用内存。



图 1.1　C 语言的优点

1.2.3　可移植性

C 是可移植的语言。这意味着，在一种系统中编写的 C 程序稍作修改或不修改就能在其他系统运行。如需修改，也只需简单更改主程序头文件中的少许项即可。大部分语言都希望成为可移植语言，但是，如果经历过把 IBM PC BASIC 程序转换成苹果 BASIC（两者是近亲），或者在 UNIX 系统中运行 IBM 大型机的 FORTRAN 程序的人都知道，移植是最麻烦的事。C 语言是可移植方面的佼佼者。从 8 位微处理器到克雷超级计算机，许多计算机体系结构都可以使用 C 编译器（C 编译器是把 C 代码转换成计算机内部指令的程序）。但是要注意，程序中针对特殊硬件设备（如，显示监视器）或操作系统特殊功能（如，Windows 8 或 OS X）编写的部分，通常是不可移植的。

由于 C 语言与 UNIX 关系密切，UNIX 系统通常会将 C 编译器作为软件包的一部分。安装 Linux 时，通常也会安装 C 编译器。供个人计算机使用的 C 编译器很多，运行各种版本的 Windows 和 Macintosh（即，Mac）的 PC 都能找到合适的 C 编译器。因此，无论是使用家庭计算机、专业工作站，还是大型机，都能找到针对特定系统的 C 编译器。

1.2.4 强大而灵活

C 语言功能强大且灵活（计算机领域经常使用这两个词）。例如，功能强大且灵活的 UNIX 操作系统，大部分是用 C 语言写的；其他语言（如，FORTRAN、Perl、Python、Pascal、LISP、Logo、BASIC）的许多编译器和解释器都是用 C 语言编写的。因此，在 UNIX 机上使用 FORTRAN 时，最终是由 C 程序生成最后的可执行程序。C 程序可以用于解决物理学和工程学的问题，甚至可用于制作电影的动画特效。

1.2.5 面向程序员

C 语言是为了满足程序员的需求而设计的，程序员利用 C 可以访问硬件、操控内存中的位。C 语言有丰富的运算符，能让程序员简洁地表达自己的意图。C 没有 Pascal 严谨，但是却比 C++的限制多。这样的灵活性既是优点也是缺点。优点是，许多任务用 C 来处理都非常简洁（如，转换数据的格式）；缺点是，你可能会犯一些莫名其妙的错误，这些错误不可能在其他语言中出现。C 语言在提供更多自由的同时，也让使用者承担了更大的责任。

另外，大多数 C 实现都有一个大型的库，包含众多有用的 C 函数。这些函数用于处理程序员经常需要解决的问题。

1.2.6 缺点

人无完人，金无足赤。C 语言也有一些缺点。例如，前面提到的，要享受用 C 语言自由编程的乐趣，就必须承担更多的责任。特别是，C 语言使用指针，而涉及指针的编程错误往往难以察觉。有句话说的好：想拥有自由就必须时刻保持警惕。

C 语言紧凑简洁，结合了大量的运算符。正因如此，我们也可以编写出让人极其费解的代码。虽然没必要强迫自己编写晦涩的代码，但是有兴趣写写也无妨。试问，除 C 语言外还为哪种语言举办过年度混乱代码大赛[1]？

瑕不掩瑜，C 语言的优点比缺点多很多。我们不想在这里多费笔墨，还是来聊聊 C 语言的其他话题。

1.3 C 语言的应用范围

早在 20 世纪 80 年代，C 语言就已经成为小型计算机（UNIX 系统）使用的主流语言。从那以后，C 语言的应用范围扩展到微型机（个人计算机）和大型机（庞然大物）。如图 1.2 所示，许多软件公司都用 C 语言来开发文字处理程序、电子表格、编译器和其他产品，因为用 C 语言编写的程序紧凑而高效。更重要的是，C 程序很方便修改，而且移植到新型号的计算机中也没什么问题。

无论是软件公司、经验丰富的 C 程序员，还是其他用户，都能从 C 语言中受益。越来越多的计算机用户已转而求助 C 语言解决一些安全问题。不一定非得是计算机专家也能使用 C 语言。

20 世纪 90 年代，许多软件公司开始改用 C++来开发大型的编程项目。C++在 C 语言的基础上嫁接了面向对象编程工具（面向对象编程是一门哲学，它通过对语言建模来适应问题，而不是对问题建模以适应语言）。C++几乎是 C 的超集，这意味着任何 C 程序差不多就是一个 C++程序。学习 C 语言，也相当于学习了许多 C++的知识。

[1] 国际 C 语言混乱代码大赛（IOCCC，The International Obfuscated C Code Contest）。这是一项国际编程赛事，从 1984 年开始，每年举办一次（1997、1999、2002、2003 和 2006 年除外），目的是写出最有创意且最让人难以理解的 C 语言代码。——译者注

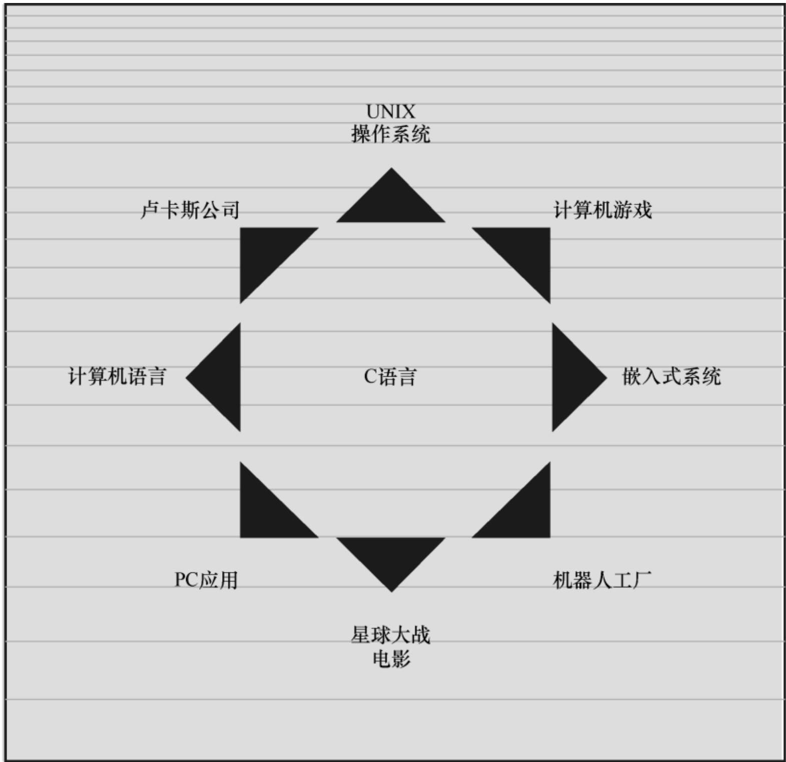


图 1.2　C 语言的应用范围

虽然这些年来 C++和 JAVA 非常流行，但是 C 语言仍是软件业中的核心技能。在最想具备的技能中，C 语言通常位居前十。特别是，C 语言已成为嵌入式系统编程的流行语言。也就是说，越来越多的汽车、照相机、DVD 播放机和其他现代化设备的微处理器都用 C 语言进行编程。除此之外，C 语言还从长期被 FORTRAN 独占的科学编程领域分得一杯羹。最终，作为开发操作系统的卓越语言，C 在 Linux 开发中扮演着极其重要的角色。因此，在进入 21 世纪的第 2 个 10 年中，C 语言仍然保持着强劲的势头。

简而言之，C 语言是最重要的编程语言之一，将来也是如此。如果你想拿下一份编程的工作，被问到是否会 C 语言时，最好回答“是”。

1.4　计算机能做什么

在学习如何用 C 语言编程之前，最好先了解一下计算机的工作原理。这些知识有助于你理解用 C 语言编写程序和运行 C 程序时所发生的事情之间有什么联系。

现代的计算机由多种部件构成。中央处理单元（CPU）承担绝大部分的运算工作。随机存取内存（RAM）是存储程序和文件的工作区；而永久内存存储设备（过去一般指机械硬盘，现在还包括固态硬盘）即使在关闭计算机后，也不会丢失之前储存的程序和文件。另外，还有各种外围设备（如，键盘、鼠标、触摸屏、监视器）提供人与计算机之间的交互。CPU 负责处理程序，接下来我们重点讨论它的工作原理。

CPU 的工作非常简单，至少从以下简短的描述中看是这样。它从内存中获取并执行一条指令，然后再从内存中获取并执行下一条指令，诸如此类（一个吉赫兹的 CPU 一秒钟能重复这样的操作大约十亿次，因此，CPU 能以惊人的速度从事枯燥的工作）。CPU 有自己的小工作区——由若干个寄存器组成，每个寄存器都可以储存一个数字。一个寄存器储存下一条指令的内存地址，CPU 使用该地址来获取和更新下一条指令。在获取指令后，CPU 在另一个寄存器中储存该指令，并更新第 1 个寄存器储存下一条指令的地址。CPU

能理解的指令有限（这些指令的集合叫作指令集）。而且，这些指令相当具体，其中的许多指令都是用于请求计算机把一个数字从一个位置移动到另一个位置。例如，从内存移动到寄存器。

下面介绍两个有趣的知识。其一，储存在计算机中的所有内容都是数字。计算机以数字形式储存数字和字符（如，在文本文档中使用的字母）。每个字符都有一个数字码。计算机载入寄存器的指令也以数字形式储存，指令集中的每条指令都有一个数字码。其二，计算机程序最终必须以数字指令码（即，机器语言）来表示。

简而言之，计算机的工作原理是：如果希望计算机做某些事，就必须为其提供特殊的指令列表（程序），确切地告诉计算机要做的事以及如何做。你必须用计算机能直接明白的语言（机器语言）创建程序。这是一项繁琐、乏味、费力的任务。计算机要完成诸如两数相加这样简单的事，就得分成类似以下几个步骤。

1．从内存位置 2000 上把一个数字拷贝到寄存器 1。

2．从内存位置 2004 上把另一个数字拷贝到寄存器 2。

3．把寄存器 2 中的内容与寄存器 1 中的内容相加，把结果储存在寄存器 1 中。

4．把寄存器 1 中的内容拷贝到内存位置 2008。

而你要做的是，必须用数字码来表示以上的每个步骤！

如果以这种方式编写程序很合你的意，那不得不说抱歉，因为用机器语言编程的黄金时代已一去不复返。但是，如果你对有趣的事情比较感兴趣，不妨试试高级编程语言。

1.5 高级计算机语言和编译器

高级编程语言（如，C）以多种方式简化了编程工作。首先，不必用数字码表示指令；其次，使用的指令更贴近你如何想这个问题，而不是类似计算机那样繁琐的步骤。使用高级编程语言，可以在更抽象的层面表达你的想法，不用考虑 CPU 在完成任务时具体需要哪些步骤。例如，对于两数相加，可以这样写：

```
total = mine + yours;
```

对我们而言，光看这行代码就知道要计算机做什么；而看用机器语言写成的等价指令（多条以数字码形式表现的指令）则费劲得多。但是，对计算机而言却恰恰相反。在计算机看来，高级指令就是一堆无法理解的无用数据。编译器在这里派上了用场。编译器是把高级语言程序翻译成计算机能理解的机器语言指令集的程序。程序员进行高级思维活动，而编译器则负责处理冗长乏味的细节工作。

编译器还有一个优势。一般而言，不同 CPU 制造商使用的指令系统和编码格式不同。例如，用 Intel Core i7（英特尔酷睿 i7）CPU 编写的机器语言程序对于 ARM Cortex-A57 CPU 而言什么都不是。但是，可以找到与特定类型 CPU 匹配的编译器。因此，使用合适的编译器或编译器集，便可把一种高级语言程序转换成供各种不同类型 CPU 使用的机器语言程序。一旦解决了一个编程问题，便可让编译器集翻译成不同 CPU 使用的机器语言。

简而言之，高级语言（如 C、Java、Pascal）以更抽象的方式描述行为，不受限于特定 CPU 或指令集。而且，高级语言简单易学，用高级语言编程比用机器语言编程容易得多。

> 1964 年，控制数据公司（Control Data Corporation）研制出了 CDC 6600 计算机。这台庞然大物是世界上首台超级计算机，当时的售价是 600 万美元。它是高能核物理研究的首选。然而，现在的普通智能手机在计算能力和内存方面都超过它数百倍，而且能看视频，放音乐。
>
> 1964 年，在工程和科学领域的主流编程语言是 FORTRAN。虽然编程语言不如硬件发展那么突飞猛进，但是也发生了很大变化。为了应对越来越大型的编程项目，语言先后为结构化编程和面向对象编程提供了更多的支持。随着时间的推移，不仅新语言层出不穷，而且现有语言也会发生变化。

1.6　语言标准

目前，有许多 C 实现可用。在理想情况下，编写 C 程序时，假设该程序中未使用机器特定的编程技术，那么它的运行情况在任何实现中都应该相同。要在实践中做到这一点，不同的实现要遵循同一个标准。

C 语言发展之初，并没有所谓的 C 标准。1987 年，布莱恩 • 柯林汉（*Brian Kernighan*）和丹尼斯 • 里奇（*Dennis Ritchie*）合著的 *The C Programming Language*（《C 语言程序设计》）第 1 版是公认的 C 标准，通常称之为 *K&R C* 或经典 *C*。特别是，该书中的附录中的“C 语言参考手册”已成为实现 C 的指导标准。例如，编译器都声称提供完整的 K&R 实现。虽然这本书中的附录定义了 C 语言，但却没有定义 C 库。与大多数语言不同的是，C 语言比其他语言更依赖库，因此需要一个标准库。实际上，由于缺乏官方标准，UNIX 实现提供的库已成为了标准库。

1.6.1　第 1 个 ANSI/ISO C 标准

随着 C 的不断发展，越来越广泛地应用于更多系统中，C 社区意识到需要一个更全面、更新颖、更严格的标准。鉴于此，美国国家标准协会（ANSI）于 1983 年组建了一个委员会（X3J11），开发了一套新标准，并于 1989 年正式公布。该标准（ANSI C）定义了 C 语言和 C 标准库。国际标准化组织于 1990 年采用了这套 C 标准（ISO C）。ISO C 和 ANSI C 是完全相同的标准。ANSI/ISO 标准的最终版本通常叫作 *C89*（因为 ANSI 于 1989 年批准该标准）或 *C90*（因为 ISO 于 1990 年批准该标准）。另外，由于 ANSI 先公布 C 标准，因此业界人士通常使用 ANSI C。

在该委员会制定的指导原则中，最有趣的可能是：保持 C 的精神。委员会在表述这一精神时列出了以下几点：

- 信任程序员；
- 不要妨碍程序员做需要做的事；
- 保持语言精练简单；
- 只提供一种方法执行一项操作；
- 让程序运行更快，即使不能保证其可移植性。

在最后一点上，标准委员会的用意是：作为实现，应该针对目标计算机来定义最合适的某特定操作，而不是强加一个抽象、统一的定义。在学习 C 语言过程中，许多方面都反映了这一哲学思想。

1.6.2　C99 标准

1994 年，ANSI/ISO 联合委员会（C9X 委员会）开始修订 C 标准，最终发布了 C99 标准。该委员会遵循了最初 C90 标准的原则，包括保持语言的精练简单。委员会的用意不是在 C 语言中添加新特性，而是为了达到新的目标。第 1 个目标是，支持国际化编程。例如，提供多种方法处理国际字符集。第 2 个目标是，“调整现有实践致力于解决明显的缺陷”。因此，在遇到需要将 C 移至 64 位处理器时，委员会根据现实生活中处理问题的经验来添加标准。第 3 个目标是，为适应科学和工程项目中的关键数值计算，提高 C 的适应性，让 C 比 FORTRAN 更有竞争力。

这 3 点（国际化、弥补缺陷和提高计算的实用性）是主要的修订目标。在其他方面的改变则更为保守，例如，尽量与 C90、C++兼容，让语言在概念上保持简单。用委员会的话说：“……委员会很满意让 C++成为大型、功能强大的语言”。

C99 的修订保留了 C 语言的精髓，C 仍是一门简洁高效的语言。本书指出了许多 C99 修改的地方。虽然该标准已发布了很长时间，但并非所有的编译器都完全实现 C99 的所有改动。因此，你可能发现 C99 的一些改动在自己的系统中不可用，或者只有改变编译器的设置才可用。

1.6.3　C11 标准

维护标准任重道远。标准委员会在 2007 年承诺 C 标准的下一个版本是 C1X，2011 年终于发布了 C11 标准。此次，委员会提出了一些新的指导原则。出于对当前编程安全的担忧，不那么强调“信任程序员”目标了。而且，供应商并未像对 C90 那样很好地接受和支持 C99。这使得 C99 的一些特性成为 C11 的可选项。因为委员会认为，不应要求服务小型机市场的供应商支持其目标环境中用不到的特性。另外需要强调的是，修订标准的原因不是因为原标准不能用，而是需要跟进新的技术。例如，新标准添加了可选项支持当前使用多处理器的计算机。对于 C11 标准，我们浅尝辄止，深入分析这部分内容已超出本书讨论的范围。

> **注意**
>
> 本书使用术语 ANSI C、ISO C 或 ANSI/ISO C 讲解 C89/90 和较新标准共有的特性，用 C99 或 C11 介绍新的特性。有时也使用 C90（例如，讨论一个特性被首次加入 C 语言时）。

1.7　使用 C 语言的 7 个步骤

C 是编译型语言。如果之前使用过编译型语言（如，Pascal 或 FORTRAN），就会很熟悉组建 C 程序的几个基本步骤。但是，如果以前使用的是解释型语言（如，BASIC）或面向图形界面语言（如，Visual Basic），或者甚至没接触过任何编程语言，就有必要学习如何编译。别担心，这并不复杂。首先，为了让读者对编程有大概的了解，我们把编写 C 程序的过程分解成 7 个步骤（见图 1.3）。注意，这是理想状态。在实际的使用过程中，尤其是在较大型的项目中，可能要做一些重复的工作，根据下一个步骤的情况来调整或改进上一个步骤。

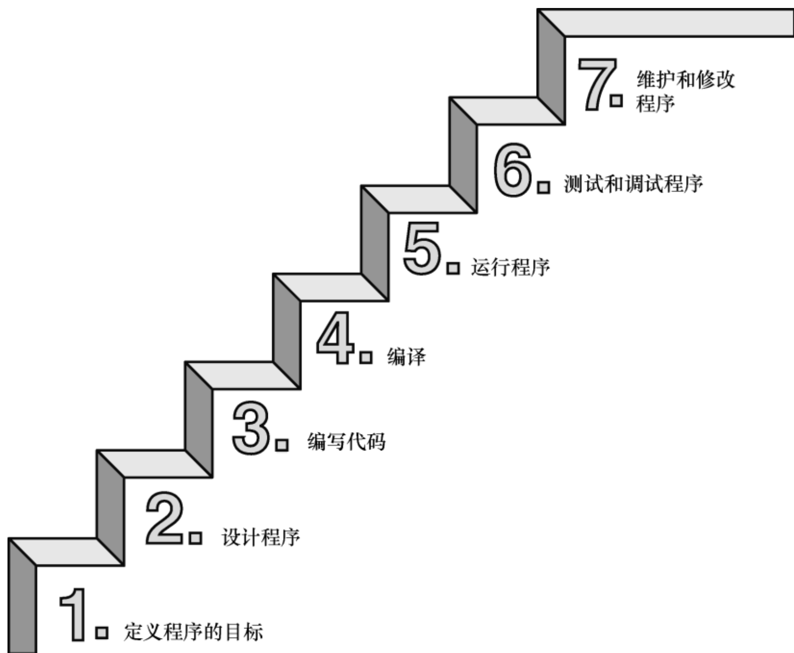


图 1.3　编程的 7 个步骤

1.7.1　第 1 步：定义程序的目标

在动手写程序之前，要在脑中有清晰的思路。想要程序去做什么首先自己要明确自己想做什么，思考你的程序需要哪些信息，要进行哪些计算和控制，以及程序应该要报告什么信息。在这一步骤中，不涉及具体的计算机语言，应该用一般术语来描述问题。

1.7.2　第 2 步：设计程序

对程序应该完成什么任务有概念性的认识后，就应该考虑如何用程序来完成它。例如，用户界面应该是怎样的？如何组织程序？目标用户是谁？准备花多长时间来完成这个程序？

除此之外，还要决定在程序（还可能是辅助文件）中如何表示数据，以及用什么方法处理数据。学习 C 语言之初，遇到的问题都很简单，没什么可选的。但是，随着要处理的情况越来越复杂，需要决策和考虑的方面也越来越多。通常，选择一个合适的方式表示信息可以更容易地设计程序和处理数据。

再次强调，应该用一般术语来描述问题，而不是用具体的代码。但是，你的某些决策可能取决于语言的特性。例如，在数据表示方面，C 的程序员就比 Pascal 的程序员有更多选择。

1.7.3　第 3 步：编写代码

设计好程序后，就可以编写代码来实现它。也就是说，把你设计的程序翻译成 C 语言。这里是真正需要使用 C 语言的地方。可以把思路写在纸上，但是最终还是要把代码输入计算机。这个过程的机制取决于编程环境，我们稍后会详细介绍一些常见的环境。一般而言，使用文本编辑器创建*源代码*文件。该文件中内容就是你翻译的 C 语言代码。程序清单 1.1 是一个 C 源代码的示例。

程序清单 1.1　C 源代码示例

```
#include <stdio.h>
int main(void)
{
    int dogs;

    printf("How many dogs do you have?\n");
    scanf("%d", &dogs);
    printf("So you have %d dog(s)!\n", dogs);

    return 0;
}
```

在这一步骤中，应该给自己编写的程序添加文字注释。最简单的方式是使用 C 的注释工具在源代码中加入对代码的解释。第 2 章将详细介绍如何在代码中添加注释。

1.7.4　第 4 步：编译

接下来的这一步是编译源代码。再次提醒读者注意，编译的细节取决于编程的环境，我们稍后马上介绍一些常见的编程环境。现在，先从概念的角度讲解编译发生了什么事情。

前面介绍过，编译器是把源代码转换成可执行代码的程序。*可执行代码*是用计算机的*机器语言*表示的代码。这种语言由数字码表示的指令组成。如前所述，不同的计算机使用不同的机器语言方案。C 编译器负责把 C 代码翻译成特定的机器语言。此外，C 编译器还将源代码与 C 库（库中包含大量的标准函数供用户使用，如 printf()和 scanf()）的代码合并成最终的程序（更精确地说，应该是由一个被称为*链接器*

的程序来链接库函数，但是在大多数系统中，编译器运行链接器）。其结果是，生成一个用户可以运行的可执行文件，其中包含着计算机能理解的代码。

编译器还会检查 C 语言程序是否有效。如果 C 编译器发现错误，就不生成可执行文件并报错。理解特定编译器报告的错误或警告信息是程序员要掌握的另一项技能。

1.7.5 第 5 步：运行程序

传统上，可执行文件是可运行的程序。在常见环境（包括 Windows 命令提示符模式、UNIX 终端模式和 Linux 终端模式）中运行程序要输入可执行文件的文件名，而其他环境可能要运行命令（如，在 VAX 中的 VMS[1]）或一些其他机制。例如，在 Windows 和 Macintosh 提供的*集成开发环境*（IDE）中，用户可以在 IDE 中通过选择菜单中的选项或按下特殊键来编辑和执行 C 程序。最终生成的程序可通过单击或双击文件名或图标直接在操作系统中运行。

1.7.6 第 6 步：测试和调试程序

程序能运行是个好迹象，但有时也可能会出现运行错误。接下来，应该检查程序是否按照你所设计的思路运行。你会发现你的程序中有一些错误，计算机行话叫作 bug。查找并修复程序错误的过程叫*调试*。学习的过程中不可避免会犯错，学习编程也是如此。因此，当你把所学的知识应用于编程时，最好为自己会犯错做好心理准备。随着你越来越老练，你所写的程序中的错误也会越来越不易察觉。

将来犯错的机会很多。你可能会犯基本的设计错误，可能错误地实现了一个好想法，可能忽视了输入检查导致程序瘫痪，可能会把圆括号放错地方，可能误用 C 语言或打错字，等等。把你将来犯错的地方列出来，这份错误列表应该会很长。

看到这里你可能会有些绝望，但是情况没那么糟。现在的编译器会捕获许多错误，而且自己也可以找到编译器未发现的错误。在学习本书的过程中，我们会给读者提供一些调试的建议。

1.7.7 第 7 步：维护和修改代码

创建完程序后，你发现程序有错，或者想扩展程序的用途，这时就要修改程序。例如，用户输入以 Zz 开头的姓名时程序出现错误、你想到了一个更好的解决方案、想添加一个更好的新特性，或者要修改程序使其能在不同的计算机系统中运行，等等。如果在编写程序时清楚地做了注释并采用了合理的设计方案，这些事情都很简单。

1.7.8 说明

编程并非像描述那样是一个线性的过程。有时，要在不同的步骤之间往复。例如，在写代码时发现之前的设计不切实际，或者想到了一个更好的解决方案，或者等程序运行后，想改变原来的设计思路。对程序做文字注释为今后的修改提供了方便。

许多初学者经常忽略第 1 步和第 2 步（定义程序目标和设计程序），直接跳到第 3 步（编写代码）。刚开始学习时，编写的程序非常简单，完全可以在脑中构思好整个过程。即使写错了，也很容易发现。但是，随着编写的程序越来越庞大、越来越复杂，动脑不动手可不行，而且程序中隐藏的错误也越来越难找。最终，那些跳过前两个步骤的人往往浪费了更多的时间，因为他们写出的程序难看、缺乏条理、让人难以理解。要编写的程序越大越复杂，事先定义和设计程序环节的工作量就越大。

---

[1] VAX（Virtual Address eXtension）是一种可支持机器语言和虚拟地址的 32 位小型计算机。VMS（Virtual Memory System）是旧名，现在叫 OpenVMS，是一种用于服务器的操作系统，可在 VAX、Alpha 或 Itanium 处理器系列平台上运行。——译者注

磨刀不误砍柴工，应该养成先规划再动手编写代码的好习惯，用纸和笔记录下程序的目标和设计框架。这样在编写代码的过程中会更加得心应手、条理清晰。

1.8　编程机制

生成程序的具体过程因计算机环境而异。C 是可移植性语言，因此可以在许多环境中使用，包括 UNIX、Linux、MS-DOS（一些人仍在使用）、Windows 和 Macintosh OS。有些产品会随着时间的推移发生演变或被取代，本书无法涵盖所有环境。

首先，来看看许多 C 环境（包括上面提到的 5 种环境）共有的一些方面。虽然不必详细了解计算机内部如何运行 C 程序，但是，了解一下编程机制不仅能丰富编程相关的背景知识，还有助于理解为何要经过一些特殊的步骤才能得到 C 程序。

用 C 语言编写程序时，编写的内容被储存在文本文件中，该文件被称为*源代码文件*（*source code file*）。大部分 C 系统，包括之前提到的，都要求文件名以.c 结尾（如，`wordcount.c` 和 `budget.c`）。在文件名中，点号（.）前面的部分称为*基本名*（*basename*），点号后面的部分称为*扩展名*（*extension*）。因此，`budget` 是基本名，`c` 是扩展名。基本名与扩展名的组合（`budget.c`）就是文件名。文件名应该满足特定计算机操作系统的特殊要求。例如，MS-DOS 是 IBM PC 及其兼容机的操作系统，比较老旧，它要求基本名不能超过 8 个字符。因此，刚才提到的文件名 `wordcount.c` 就是无效的 DOS 文件名。有些 UNIX 系统限制整个文件名（包括扩展名）不超过 14 个字符，而有些 UNIX 系统则允许使用更长的文件名，最多 255 个字符。Linux、Windows 和 Macintosh OS 都允许使用长文件名。

接下来，我们来看一下具体的应用，假设有一个名为 `concrete.c` 的源文件，其中的 C 源代码如程序清单 1.2 所示。

程序清单 1.2　c 程序

```
#include <stdio.h>
int main(void)
{
    printf("Concrete contains gravel and cement.\n");

    return 0;
}
```

如果看不懂程序清单 1.2 中的代码，不用担心，我们将在第 2 章学习相关知识。

1.8.1　目标代码文件、可执行文件和库

C 编程的基本策略是，用程序把源代码文件转换为可执行文件（其中包含可直接运行的机器语言代码）。典型的 C 实现通过编译和链接两个步骤来完成这一过程。编译器把源代码转换成中间代码，链接器把中间代码和其他代码合并，生成可执行文件。C 使用这种分而治之的方法方便对程序进行模块化，可以独立编译单独的模块，稍后再用链接器合并已编译的模块。通过这种方式，如果只更改某个模块，不必因此重新编译其他模块。另外，链接器还将你编写的程序和预编译的库代码合并。

中间文件有多种形式。我们在这里描述的是最普遍的一种形式，即把源代码转换为机器语言代码，并把结果放在*目标代码文件*（或简称*目标文件*）中（这里假设源代码只有一个文件）。虽然目标文件中包含机器语言代码，但是并不能直接运行该文件。因为目标文件中储存的是编译器翻译的源代码，这还不是一个完整的程序。

目标代码文件缺失*启动代码*（*startup code*）。启动代码充当着程序和操作系统之间的接口。例如，可以在 MS Windows 或 Linux 系统下运行 IBM PC 兼容机。这两种情况所使用的硬件相同，所以目标代码相同，

但是 Windows 和 Linux 所需的启动代码不同，因为这些系统处理程序的方式不同。

目标代码还缺少库函数。几乎所有的 C 程序都要使用 C 标准库中的函数。例如，`concrete.c` 中就使用了 `printf()` 函数。目标代码文件并不包含该函数的代码，它只包含了使用 `printf()` 函数的指令。`printf()` 函数真正的代码储存在另一个被称为库的文件中。库文件中有许多函数的目标代码。

链接器的作用是，把你编写的目标代码、系统的标准启动代码和库代码这 3 部分合并成一个文件，即可执行文件。对于库代码，链接器只会把程序中要用到的库函数代码提取出来（见图 1.4）。

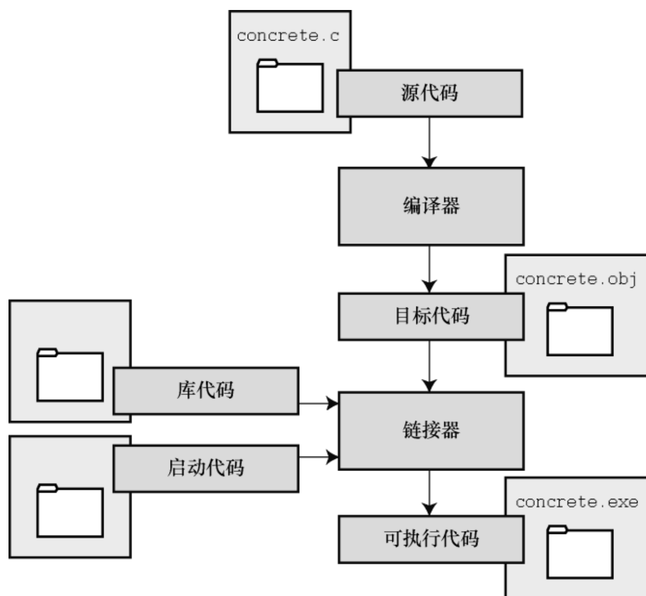


图 1.4　编译器和链接器

简而言之，目标文件和可执行文件都由机器语言指令组成的。然而，目标文件中只包含编译器为你编写的代码翻译的机器语言代码，可执行文件中还包含你编写的程序中使用的库函数和启动代码的机器代码。

在有些系统中，必须分别运行编译程序和链接程序，而在另一些系统中，编译器会自动启动链接器，用户只需给出编译命令即可。

接下来，了解一些具体的系统。

1.8.2　UNIX 系统

由于 C 语言因 UNIX 系统而生，也因此而流行，所以我们从 UNIX 系统开始（注意：我们提到的 UNIX 还包含其他系统，如 FreeBSD，它是 UNIX 的一个分支，但是由于法律原因不使用该名称）。

1.　在 UNIX 系统上编辑

UNIX C 没有自己的编辑器，但是可以使用通用的 UNIX 编辑器，如 emacs、jove、vi 或 X Window System 文本编辑器。

作为程序员，要负责输入正确的程序和为储存该程序的文件起一个合适的文件名。如前所述，文件名应该以`.c` 结尾。注意，UNIX 区分大小写。因此，`budget.c`、`BUDGET.c` 和 `Budget.c` 是 3 个不同但都有效的 C 源文件名。但是 `BUDGET.C` 是无效文件名，因为该名称的扩展名使用了大写 `C` 而不是小写 `c`。

假设我们在 vi 编译器中编写了下面的程序，并将其储存在 `inform.c` 文件中：

```
#include <stdio.h>
int main(void)
```

```
{
    printf("A .c is used to end a C program filename.\n");

    return 0;
}
```

以上文本就是源代码，inform.c 是源文件。注意，源文件是整个编译过程的开始，不是结束。

2. 在 UNIX 系统上编译

虽然在我们看来，程序完美无缺，但是对计算机而言，这是一堆乱码。计算机不明白#include 和 printf 是什么（也许你现在也不明白，但是学到后面就会明白，而计算机却不会）。如前所述，我们需要编译器将我们编写的代码（源代码）翻译成计算机能看懂的代码（机器代码）。最后生成的可执行文件中包含计算机要完成任务所需的所有机器代码。

以前，UNIX C 编译器要调用语言定义的 cc 命令。但是，它没有跟上标准发展的脚步，已经退出了历史舞台。但是，UNIX 系统提供的 C 编译器通常来自一些其他源，然后以 cc 命令作为编译器的别名。因此，虽然在不同的系统中会调用不同的编译器，但用户仍可以继续使用相同的命令。

编译 inform.c，要输入以下命令：

```
cc inform.c
```

几秒钟后，会返回 UNIX 的提示，告诉用户任务已完成。如果程序编写错误，你可能会看到警告或错误消息，但我们先假设编写的程序完全正确（如果编译器报告 void 的错误，说明你的系统未更新成 ANSI C 编译器，只需删除 void 即可）。如果使用 ls 命令列出文件，会发现有一个 a.out 文件（见图 1.5）。该文件是包含已翻译（或已编译）程序的可执行文件。要运行该文件，只需输入：

```
a.out
```

输出内容如下：

```
A .c is used to end a C program filename.
```

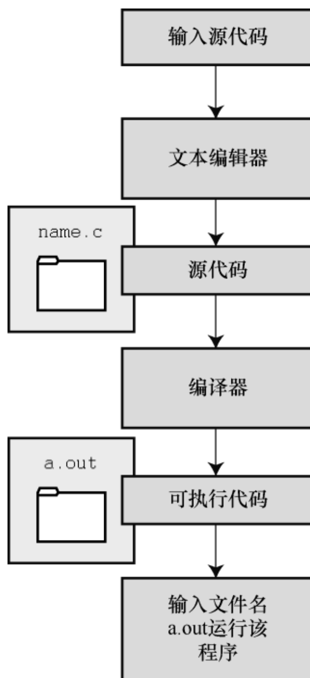


图 1.5　用 UNIX 准备 C 程序

如果要储存可执行文件（a.out），应该把它重命名。否则，该文件会被下一次编译程序时生成的新 a.out 文件替换。

如何处理目标代码？C 编译器会创建一个与源代码基本名相同的目标代码文件，但是其扩展名是.o。在该例中，目标代码文件是 inform.o。然而，却找不到这个文件，因为一旦链接器生成了完整的可执行程序，就会将其删除。如果原始程序有多个源代码文件，则保留目标代码文件。学到后面多文件程序时，你会明白到这样做的好处。

1.8.3　GNU 编译器集合和 LLVM 项目

GNU 项目始于 1987 年，是一个开发大量免费 UNIX 软件的集合（GNU 的意思是“GNU’s Not UNIX”，即 GNU 不是 UNIX）。GNU 编译器集合（也被称为 GCC，其中包含 GCC C 编译器）是该项目的产品之一。GCC 在一个指导委员会的带领下，持续不断地开发，它的 C 编译器紧跟 C 标准的改动。GCC 有各种版本以适应不同的硬件平台和操作系统，包括 UNIX、Linux 和 Windows。用 gcc 命令便可调用 GCC C 编译器。许多使用 gcc 的系统都用 cc 作为 gcc 的别名。

LLVM 项目成为 cc 的另一个替代品。该项目是与编译器相关的开源软件集合，始于伊利诺伊大学的 2000 份研究项目。它的 Clang 编译器处理 C 代码，可以通过 clang 调用。有多种版本供不同的平台使用，包括 Linux。2012 年，Clang 成为 FreeBSD 的默认 C 编译器。Clang 也对最新的 C 标准支持得很好。

GNU 和 LLVM 都可以使用-v 选项来显示版本信息，因此各系统都使用 cc 别名来代替 gcc 或 clang 命令。以下组合：

```
cc -v
```

显示你所使用的编译器及其版本。

gcc 和 clang 命令都可以根据不同的版本选择运行时选项来调用不同 C 标准。

```
gcc -std=c99 inform.c[1]
gcc -std=c1x inform.c
gcc -std=c11 inform.c
```

第 1 行调用 C99 标准，第 2 行调用 GCC 接受 C11 之前的草案标准，第 3 行调用 GCC 接受的 C11 标准版本。Clang 编译器在这一点上用法与 GCC 相同。

1.8.4　Linux 系统

Linux 是一个开源、流行、类似于 UNIX 的操作系统，可在不同平台（包括 PC 和 Mac）上运行。在 Linux 中准备 C 程序与在 UNIX 系统中几乎一样，不同的是要使用 GNU 提供的 GCC 公共域 C 编译器。编译命令类似于：

```
gcc inform.c
```

注意，在安装 Linux 时，可选择是否安装 GCC。如果之前没有安装 GCC，则必须安装。通常，安装过程会将 cc 作为 gcc 的别名，因此可以在命令行中使用 cc 来代替 gcc。

欲详细了解 GCC 和最新发布的版本，请访问 http://www.gnu.org/software/gcc/index.html。

[1] GCC 最基本的用法是：gcc [options] [filenames]，其中 options 是所需的参数，filenames 是文件名。——译者注

1.8.5　PC 的命令行编译器

C 编译器不是标准 Windows 软件包的一部分，因此需要从别处获取并安装 C 编译器。可以从互联网免费下载 Cygwin 和 MinGW，这样便可在 PC 上通过命令行使用 GCC 编译器。Cygwin 在自己的视窗运行，模仿 Linux 命令行环境，有一行命令提示。MinGW 在 Windows 的命令提示模式中运行。这和 GCC 的最新版本一样，支持 C99 和 C11 最新的一些功能。Borland 的 C++编译器 5.5 也可以免费下载，支持 C90。

源代码文件应该是文本文件，不是字处理器文件（字处理器文件包含许多额外的信息，如字体和格式等）。因此，要使用文本编辑器（如，Windows Notepad）来编辑源代码。如果使用字处理器，要以文本模式另存文件。源代码文件的扩展名应该是`.c`。一些字处理器会为文本文件自动添加`.txt` 扩展名。如果出现这种情况，要更改文件名，把 `txt` 替换成 `c`。

通常，C 编译器生成的中间目标代码文件的扩展名是`.obj`（也可能是其他扩展名）。与 UNIX 编译器不同，这些编译器在完成编译后通常不会删除这些中间文件。有些编译器生成带`.asm` 扩展名的汇编语言文件，而有些编译器则使用自己特有的格式。

一些编译器在编译后会自动运行链接器，另一些要求用户手动运行链接器。在可执行文件中链接的结果是，在原始的源代码基本名后面加上`.exe` 扩展名。例如，编译和链接 `concrete.c` 源代码文件，生成的是 `concrete.exe` 文件。可以在命令行输入基本名来运行该程序：

```
C>concrete
```

1.8.6　集成开发环境（Windows）

许多供应商（包括微软、Embarcadero、Digital Mars）都提供 Windows 下的集成开发环境，或称为 IDE（目前，大多数 IDE 都是 C 和 C++结合的编译器）。可以免费下载的 IDE 有 Microsoft Visual Studio Express 和 Pelles C。利用集成开发环境可以快速开发 C 程序。关键是，这些 IDE 都内置了用于编写 C 程序的编辑器。这类集成开发环境都提供了各种菜单（如，命名、保存源代码文件、编译程序、运行程序等），用户不用离开 IDE 就能顺利编写、编译和运行程序。如果编译器发现错误，会返回编辑器中，标出有错误的行号，并简单描述情况。

初次接触 Windows IDE 可能会望而生畏，因为它提供了多种*目标*（*target*），即运行程序的多种环境。例如，IDE 提供了 32 位 Windows 程序、64 位 Windows 程序、动态链接库文件（DLL）等。许多目标都涉及 Windows 图形界面。要管理这些（及其他）选择，通常要先创建一个*项目*（*project*），以便稍后在其中添加待使用的源代码文件名。不同的产品具体步骤不同。一般而言，首先使用【文件】菜单或【项目】菜单创建一个项目。选择正确的项目形式非常重要。本书中的例子都是一般示例，针对在简单的命令行环境中运行而设计。Windows IDE 提供多种选择以满足用户的不同需求。例如，Microsoft Visual Studio 提供【Win32 控制台应用程序】选项。对于其他系统，查找一个诸如【DOS EXE】、【Console】或【Character Mode】的可执行选项。选择这些模式后，将在一个类控制台窗口中运行可执行程序。选择好正确的项目类型后，使用 IDE 的菜单打开一个新的源代码文件。对于大多数产品而言，使用【文件】菜单就能完成。你可能需要其他步骤将源文件添加到项目中。

通常，Windows IDE 既可处理 C 也可处理 C++，因此要指定待处理的程序是 C 还是 C++。有些产品用项目类型来区分两者，有些产品（如，Microsoft Visual C++）用`.c` 文件扩展名来指明使用 C 而不是 C++。当然，大多数 C 程序也可以作为 C++程序运行。欲了解 C 和 C++的区别，请参阅参考资料 IX。

你可能会遇到一个问题：在程序执行完毕后，执行程序的窗口立即消失。如果不希望出现这种情况，可以让程序暂停，直到按下 Enter 键，窗口才消失。要实现这种效果，可以在程序的最后（`return` 这行代码之前）添加下面一行代码：

```
getchar();
```

该行读取一次键的按下，所以程序在用户按下 Enter 键之前会暂停。有时根据程序的需要，可能还需要一个击键等待。这种情况下，必须用两次 getchar()：

```
getchar();
getchar();
```

例如，程序在最后提示用户输入体重。用户键入体重后，按下 Enter 键以输入数据。程序将读取体重，第 1 个 getchar()读取 Enter 键，第 2 个 getchar()会导致程序暂停，直至用户再次按下 Enter 键。如果你现在不知所云，没关系，在学完 C 输出后就会明白。到时，我们会提醒读者使用这种方法。

虽然许多 IDE 在使用上大体一致，但是细节上有所不同。就一个产品的系列而言，不同版本也是如此。要经过一段时间的实践，才会熟悉编译器的工作方式。必要时，还需阅读使用手册或网上教程。

Microsoft Visual Studio 和 C 标准

在 Windows 软件开发中，Microsoft Visual Studio 及其免费版本 Microsoft Visual Studio Express 都久负盛名，它们与 C 标准的关系也很重要。然而，微软鼓励程序员从 C 转向 C++和 C#。虽然 Visual Studio 支持 C89/90，但是到目前为止，它只选择性地支持那些在 C++新特性中能找到的 C 标准（如，long long 类型）。而且，自 2012 版本起，Visual Studio 不再把 C 作为项目类型的选项。尽管如此，本书中的绝大多数程序仍可用 Visual Studio 来编译。在新建项目时，选择 C++选项，然后选择【Win32 控制台应用程序】，在应用设置中选择【空项目】。几乎所有的 C 程序都能与 C++程序兼容。所以，本书中的绝大多数 C 程序都可作为 C++程序运行。或者，在选择 C++选项后，将默认的源文件扩展名.cpp 替换成.c，编译器便会使用 C 语言的规则代替 C++。

1.8.7 Windows/Linux

许多 Linux 发行版都可以安装在 Windows 系统中，以创建双系统。一些存储器会为 Linux 系统预留空间，以便可以启动 Windows 或 Linux。可以在 Windows 系统中运行 Linux 程序，或在 Linux 系统中运行 Windows 程序。不能通过 Windows 系统访问 Linux 文件，但是可以通过 Linux 系统访问 Windows 文档。

1.8.8 Macintosh 中的 C

目前，苹果免费提供 Xcode 开发系统下载（过去，它有时免费，有时付费）。它允许用户选择不同的编程语言，包括 C 语言。

Xcode 凭借可处理多种编程语言的能力，可用于多平台，开发超大型的项目。但是，首先要学会如何编写简单的 C 程序。在 Xcode 4.6 中，通过【File】菜单选择【New Project】，然后选择【OS X Application Command Line Tool】，接着输入产品名并选择 C 类型。Xcode 使用 Clang 或 GCC C 编译器来编译 C 代码，它以前默认使用 GCC，但是现在默认使用 Clang。可以设置选择使用哪一个编译器和哪一套 C 标准（因为许可方面的事宜，Xcode 中 Clang 的版本比 GCC 的版本要新）。

UNIX 系统内置 Mac OS X，终端工具打开的窗口是让用户在 UNIX 命令行环境中运行程序。苹果在标准软件包中不提供命令行编译器，但是，如果下载了 Xcode，还可以下载可选的命令行工具，这样就可以使用 clang 和 gcc 命令在命令行模式中编译。

1.9 本书的组织结构

本书采用多种方式编排内容，其中最直接的方法是介绍 A 主题的所有内容、介绍 B 主题的所有内容，

等等。这对参考类书籍来说尤为重要，读者可以在同一处找到与主题相关的所有内容。但是，这通常不是学习的最佳顺序。例如，如果在开始学习英语时，先学完所有的名词，那你的表达能力一定很有限。虽然可以指着物品说出名称，但是，如果稍微学习一些名词、动词、形容词等，再学习一些造句规则，那么你的表达能力一定会大幅提高。

为了让读者更好地吸收知识，本书采用螺旋式方法，先在前几个章节中介绍一些主题，在后面章节再详细讨论相关内容。例如，对学习 C 语言而言，理解函数至关重要。因此，我们在前几个章节中安排一些与函数相关的内容，等读者学到第 9 章时，已对函数有所了解，学习使用函数会更加容易。与此类似，前几章还概述了一些字符串和循环的内容。这样，读者在完全弄懂这些内容之前，就可以在自己的程序中使用这些有用的工具。

1.10　本书的约定

在学习 C 语言之前，先介绍一下本书的格式。

1.10.1　字体

本书用类似在屏幕上或打印输出时的字体（一种等宽字体），表示文本程序和计算机输入、输出。前面已经出现了多次，如果读者没有注意到，字体如下所示：

```
#include <stdio.h>
int main(void)
{
    printf("Concrete contains gravel and cement.\n");

    return 0;
}
```

在涉及与代码相关的术语时，也使用相同的等宽字体，如 `stdio.h`。本书用等宽斜体表示占位符，可以用具体的项替换这些占位符。例如，下面是一个声明的模型：

type_name variable_name;

这里，可用 `int` 替换 *type_name*，用 `zebra_count` 替换 *variable_name*。

1.10.2　程序输出

本书用相同的字体表示计算机的输出，粗体表示用户输入。例如，下面是第 14 章中一个程序的输出：

```
Please enter the book title.
Press [enter] at the start of a line to stop.

My Life as a Budgie
Now enter the author.

Mack Zackles
```

如上所示，以标准计算机字体显示的行表示程序的输出，粗体行表示用户的输入。

可以通过多种方式与计算机交互。在这里，我们假设读者使用键盘键入内容，在屏幕上阅读计算机的响应。

1.　特殊的击键

通常，通过按下标有 Enter、c/r、Return 或一些其他文字的键来发送指令。本书将这些按键统一称为 Enter 键。一般情况下，我们默认你在每行输入的末尾都会按下 Enter 键。尽管如此，为了标示一些特

定的位置，本书使用`[enter]`显式标出 Enter 键。方括号表示按下一次 Enter 键，而不是输入 `enter`。

除此之外，书中还会提到控制字符（如，Ctrl+D）。这种写法的意思是，在按下 Ctrl 键（也可能是 Control 键）的同时按下 D 键。

2. 本书使用的系统

C 语言的某些方面（如，储存数字的空间大小）因系统而异。本书在示例中提到“我们的系统”时，通常是指在 iMac 上运行 OS X 10.8.4，使用 Xcode 4.6.2 开发系统的 Clang 3.2 编译器。本书的大部分程序都能使用 Windows7 系统的 Microsoft Visual Studio Express 2012 和 Pelles C 7.0，以及 Ubuntu13.04 Linux 系统的 GCC 4.7.3 进行编译。

3. 读者的系统

你需要一个 C 编译器或访问一个 C 编译器。C 程序可以在多种计算机系统中运行，因此你的选择面很广。确保你使用的 C 编译器与当前使用的计算机系统匹配。本书中，除了某些示例要求编译器支持 C99 或 C11 标准，其余大部分示例都可在 C90 编译器中运行。如果你使用的编译器是早于 ANSI/ISO 的老式编译器，在编译时肯定要经常调整，很不方便。与其如此，不如换个新的编译器。

大部分编译器供应商都为学生和教学人员提供特惠版本，详情请查看供应商的网站。

1.10.3 特殊元素

本书包含一些强调特定知识点的特殊元素，提示、注意、警告，将以如下形式出现在本书中：

> **边栏**
>
> 边栏提供更深入的讨论或额外的背景，有助于解释当前的主题。

> **提示**
>
> 提示一般都短小精悍，帮助读者理解一些特殊的编程情况。

> **警告**
>
> 用于警告读者注意一些潜在的陷阱。

> **注意**
>
> 提供一些评论，提醒读者不要误入歧途。

1.11 本章小结

C 是强大而简洁的编程语言。它之所以流行，在于自身提供大量的实用编程工具，能很好地控制硬件。而且，与大多数其他程序相比，C 程序更容易从一个系统移植到另一个系统。

C 是编译型语言。C 编译器和链接器是把 C 语言源代码转换成可执行代码的程序。

用 C 语言编程可能费力、困难，让你感到沮丧，但是它也可以激发你的兴趣，让你兴奋、满意。我们希望你在愉快的学习过程中爱上 C。

1.12　复习题

复习题的参考答案在附录 A 中。

1．对编程而言，可移植性意味着什么？

2．解释源代码文件、目标代码文件和可执行文件有什么区别？

3．编程的 7 个主要步骤是什么？

4．编译器的任务是什么？

5．链接器的任务是什么？

1.13　编程练习

我们尚未要求你编写 C 代码，该练习侧重于编程过程的早期步骤。

1．你刚被 MacroMuscle 有限公司聘用。该公司准备进入欧洲市场，需要一个把英寸单位转换为厘米单位（1 英寸=2.54 厘米）的程序。该程序要提示用户输入英寸值。你的任务是定义程序目标和设计程序（编程过程的第 1 步和第 2 步）。

第 2 章

C 语言概述

本章介绍以下内容：

- 运算符：=
- 函数：main()、printf()
- 编写一个简单的 C 程序
- 创建整型变量，为其赋值并在屏幕上显示其值
- 换行字符
- 如何在程序中写注释，创建包含多个函数的程序，发现程序的错误
- 什么是关键字

C 程序是什么样子的？浏览本书，能看到许多示例。初见 C 程序会觉得有些古怪，程序中有许多{、cp->tort 和*ptr++这样的符号。然而，在学习 C 的过程中，对这些符号和 C 语言特有的其他符号会越来越熟悉，甚至会喜欢上它们。如果熟悉与 C 相关的其他语言，会对 C 语言有似曾相识的感觉。本章，我们从演示一个简单的程序示例开始，解释该程序的功能。同时，强调一些 C 语言的基本特性。

2.1 简单的 C 程序示例

我们来看一个简单的 C 程序，如程序清单 2.1 所示。该程序演示了用 C 语言编程的一些基本特性。请先通读程序清单 2.1，看看自己是否能明白该程序的用途，再认真阅读后面的解释。

程序清单 2.1 first.c **程序**

```
#include <stdio.h>
int main(void)                  /* 一个简单的 C 程序 */
{
    int num;                    /* 定义一个名为 num 的变量 */
    num = 1;                    /* 为 num 赋一个值 */

    printf("I am a simple ");   /* 使用 printf()函数 */
    printf("computer.\n");
    printf("My favorite number is %d because it is first.\n",num);

    return 0;
}
```

如果你认为该程序会在屏幕上打印一些内容，那就对了！光看程序也许并不知道打印的具体内容，所以，运行该程序，并查看结果。首先，用你熟悉的编辑器（或者编译器提供的编辑器）创建一个包含程序清单 2.1 中所有内容的文件。给该文件命名，并以.c 作为扩展名，以满足当前系统对文件名的要求。例如，可以使用 first.c。现在，编译并运行该程序（查看第 1 章，复习该步骤的具体内容）。如果一切运行正常，该程序的输出应该是：

```
I am a simple computer.
My favorite number is 1 because it is first.
```

总而言之，结果在意料之中，但是程序中的\n 和%d 是什么？程序中有几行代码看起来有点奇怪。接下来，我们逐行解释这个程序。

程序调整

程序的输出是否在屏幕上一闪而过？某些窗口环境会在单独的窗口运行程序，然后在程序运行结束后自动关闭窗口。如果遇到这种情况，可以在程序中添加额外的代码，让窗口等待用户按下一个键后才关闭。一种方法是，在程序的 return 语句前添加一行代码：

```
getchar();
```

这行代码会让程序等待击键，窗口会在用户按下一个键后才关闭。在第 8 章中会详细介绍 getchar() 的内容。

2.2　示例解释

我们会把程序清单 2.1 的程序分析两遍。第 1 遍（快速概要）概述程序中每行代码的作用，帮助读者初步了解程序。第 2 遍（程序细节）详细分析代码的具体含义，帮助读者深入理解程序。

图 2.1 总结了组成 C 程序的几个部分[1]，图中包含的元素比第 1 个程序多。

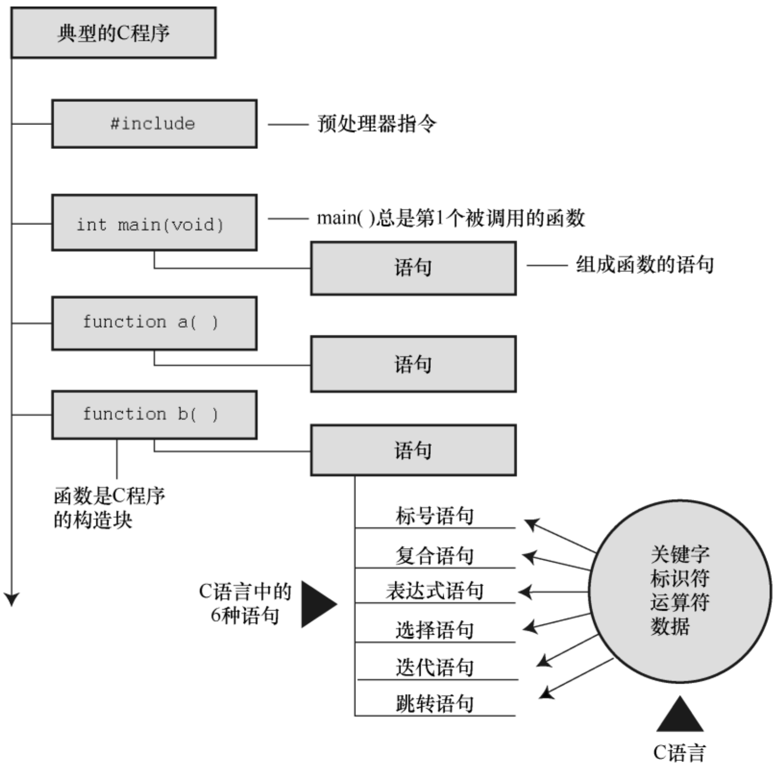


图 2.1　C 程序解剖

[1] 原书图中叙述有误。根据 C11 标准，C 语言有 6 种语句，已在图中更正。——译者注

2.2.1 第1遍：快速概要

本节简述程序中的每行代码的作用。下一节详细讨论代码的含义。

```
#include<stdio.h>        ←包含另一个文件
```

该行告诉编译器把 stdio.h 中的内容包含在当前程序中。stdio.h 是 C 编译器软件包的标准部分，它提供键盘输入和屏幕输出的支持。

```
int main(void)           ←函数名
```

C 程序包含一个或多个函数，它们是 C 程序的基本模块。程序清单 2.1 的程序中有一个名为 main()的函数。圆括号表明 main()是一个函数名。int 表明 main()函数返回一个整数，void 表明 main()不带任何参数。这些内容我们稍后详述。现在，只需记住 int 和 void 是标准 ANSI C 定义 main()的一部分（如果使用 ANSI C 之前的编译器，请省略 void；考虑到兼容的问题，请尽量使用较新的 C 编译器）。

```
/* 一个简单的C程序 */        ←注释
```

注释在/*和*/两个符号之间，这些注释能提高程序的可读性。注意，注释只是为了帮助读者理解程序，编译器会忽略它们。

```
{           ←函数体开始
```

左花括号表示函数定义开始，右花括号（}）表示函数定义结束。

```
int num;        ←声明
```

该声明表明，将使用一个名为 num 的变量，而且 num 是 int（整数）类型。

```
num = 1;        ←赋值表达式语句
```

语句 num = 1;把值 1 赋给名为 num 的变量。

```
printf("I am a simple "); ←调用一个函数
```

该语句使用 printf()函数，在屏幕上显示 I am a simple，光标停在同一行。printf()是标准的 C 库函数。在程序中使用函数叫作*调用函数*。

```
printf("computer.\n");     ←调用另一个函数
```

接下来调用的这个 printf()函数在上条语句打印出来的内容后面加上“computer”。代码\n 告诉计算机另起一行，即把光标移至下一行。

```
printf("My favorite number is %d because it is first.\n", num);
```

最后调用的 printf()把 num 的值（1）内嵌在用双引号括起来的内容中一并打印。%d 告诉计算机以何种形式输出 num 的值，打印在何处。

```
return 0;       ←return 语句
```

C 函数可以给调用方提供（或返回）一个数。目前，可暂时把该行看作是结束 main()函数的要求。

```
}           ←结束
```

必须以右花括号表示程序结束。

2.2.2 第2遍：程序细节

浏览完程序清单 2.1 后，我们来仔细分析这个程序。再次强调，本节将逐行分析程序中的代码，以每行代码为出发点，深入分析代码背后的细节，为更全面地学习 C 语言编程的特性夯实基础。

1. #include 指令和头文件

```
#include<stdio.h>
```

这是程序的第 1 行。#include <stdio.h>的作用相当于把 stdio.h 文件中的所有内容都输入该行

所在的位置。实际上，这是一种“拷贝-粘贴”的操作。include 文件提供了一种方便的途径共享许多程序共有的信息。

#include 这行代码是一条 C 预处理器指令（*preprocessor directive*）。通常，C 编译器在编译前会对源代码做一些准备工作，即预处理（*preprocessing*）。

所有的 C 编译器软件包都提供 stdio.h 文件。该文件中包含了供编译器使用的输入和输出函数（如，printf()）信息。该文件名的含义是标准输入/输出头文件。通常，在 C 程序顶部的信息集合被称为头文件（*header*）。

在大多数情况下，头文件包含了编译器创建最终可执行程序要用到的信息。例如，头文件中可以定义一些常量，或者指明函数名以及如何使用它们。但是，函数的实际代码在一个预编译代码的库文件中。简而言之，头文件帮助编译器把你的程序正确地组合在一起。

ANSI/ISO C 规定了 C 编译器必须提供哪些头文件。有些程序要包含 stdio.h，而有些不用。特定 C 实现的文档中应该包含对 C 库函数的说明。这些说明确定了使用哪些函数需要包含哪些头文件。例如，要使用 printf()函数，必须包含 stdio.h 头文件。省略必要的头文件可能不会影响某一特定程序，但是最好不要这样做。本书每次用到库函数，都会用#include 指令包含 ANSI/ISO 标准指定的头文件。

注意　为何不内置输入和输出

读者一定很好奇，为何不把输入和输出这些基本功能内置在语言中。原因之一是，并非所有的程序都会用到 I/O（输入/输出）包。轻装上阵表现了 C 语言的哲学。正是这种经济使用资源的原则，使得 C 语言成为流行的嵌入式编程语言（例如，编写控制汽车自动燃油系统或蓝光播放机芯片的代码）。#include 中的#符号表明，C 预处理器在编译器接手之前处理这条指令。本书后面章节中会介绍更多预处理器指令的示例，第 16 章将更详细地讨论相关内容。

2. main()函数

```
int main(void);
```

程序清单 2.1 中的第 2 行表明该函数名为 main。的确，main 是一个极其普通的名称，但是这是唯一的选择。C 程序一定从 main()函数开始执行（目前不必考虑例外的情况）。除了 main()函数，你可以任意命名其他函数，而且 main()函数必须是开始的函数。圆括号有什么功能？用于识别 main()是一个函数。很快你将学到更多的函数。就目前而言，只需记住函数是 C 程序的基本模块。

int 是 main()函数的返回类型。这表明 main()函数返回的值是整数。返回到哪里？返回给操作系统。我们将在第 6 章中再来探讨这个问题。

通常，函数名后面的圆括号中包含一些传入函数的信息。该例中没有传递任何信息。因此，圆括号内是单词 void（第 11 章将介绍把信息从 main()函数传回操作系统的另一种形式）。

如果浏览旧式的 C 代码，会发现程序以如下形式开始：

```
main()
```

C90 标准勉强接受这种形式，但是 C99 和 C11 标准不允许这样写。因此，即使你使用的编译器允许，也不要这样写。

你还会看到下面这种形式：

```
void main()
```

一些编译器允许这样写，但是所有的标准都未认可这种写法。因此，编译器不必接受这种形式，而且许多编译器都不能这样写。需要强调的是，只要坚持使用标准形式，把程序从一个编译器移至另一个编译

器时就不会出什么问题。

3. 注释

```
/*一个简单的程序*/
```

在程序中，被/**/两个符号括起来的部分是程序的注释。写注释能让他人（包括自己）更容易明白你所写的程序。C 语言注释的好处之一是，可将注释放在任意的地方，甚至是与要解释的内容在同一行。较长的注释可单独放一行或多行。在/*和*/之间的内容都会被编译器忽略。下面列出了一些有效和无效的注释形式：

```
/* 这是一条C注释。 */
/* 这也是一条注释，
   被分成两行。*/
/*
   也可以这样写注释。
*/

/* 这条注释无效，因为缺少了结束标记。
```

C99 新增了另一种风格的注释，普遍用于 C++和 Java。这种新风格使用//符号创建注释，仅限于单行。

```
// 这种注释只能写成一行。
int rigue; // 这种注释也可置于此。
```

因为一行末尾就标志着注释的结束，所以这种风格的注释只需在注释开始处标明//符号即可。

这种新形式的注释是为了解决旧形式注释存在的潜在问题。假设有下面的代码：

```
/*
   希望能运行。
*/
x = 100;
y = 200;
/* 其他内容已省略。 */
```

接下来，假设你决定删除第 4 行，但不小心删掉了第 3 行（*/）。代码如下所示：

```
/*
   希望能运行。
y = 200;
/*其他内容已省略。 */
```

现在，编译器把第 1 行的/*和第 4 行的*/配对，导致 4 行代码全都成了注释（包括应作为代码的那一行）。而//形式的注释只对单行有效，不会导致这种“消失代码”的问题。

一些编译器可能不支持这一特性。还有一些编译器需要更改设置，才能支持 C99 或 C11 的特性。

考虑到只用一种注释风格过于死板乏味，本书在示例中采用两种风格的注释。

4. 花括号、函数体和块

```
{
    ...
}
```

程序清单 2.1 中，花括号把 `main()` 函数括起来。一般而言，所有的 C 函数都使用花括号标记函数体的开始和结束。这是规定，不能省略。只有花括号（`{}`）能起这种作用，圆括号（`()`）和方括号（`[]`）都不行。

花括号还可用于把函数中的多条语句合并为一个单元或块。如果读者熟悉 Pascal、ADA、Modula-2 或

者 Algol，就会明白花括号在 C 语言中的作用类似于这些语言中的 begin 和 end。

5. 声明

```
int num;
```

程序清单 2.1 中，这行代码叫作声明（*declaration*）。声明是 C 语言最重要的特性之一。在该例中，声明完成了两件事。其一，在函数中有一个名为 num 的变量（*variable*）。其二，int 表明 num 是一个整数（即，没有小数点或小数部分的数）。int 是一种数据类型。编译器使用这些信息为 num 变量在内存中分配存储空间。分号在 C 语言中是大部分语句和声明的一部分，不像在 Pascal 中只是语句间的分隔符。

int 是 C 语言的一个关键字（*keyword*），表示一种基本的 C 语言数据类型。关键字是语言定义的单词，不能做其他用途。例如，不能用 int 作为函数名和变量名。但是，这些关键字在该语言以外不起作用，所以把一只猫或一个可爱的小孩叫 int 是可以的（尽管某些地方的当地习俗或法律可能不允许）。

示例中的 num 是一个标识符（*identifier*），也就一个变量、函数或其他实体的名称。因此，声明把特定标识符与计算机内存中的特定位置联系起来，同时也确定了储存在某位置的信息类型或数据类型。

在 C 语言中，所有变量都必须先声明才能使用。这意味着必须列出程序中用到的所有变量名及其类型。

以前的 C 语言，还要求把变量声明在块的顶部，其他语句不能在任何声明的前面。也就是说，main() 函数体如下所示：

```
int main() //旧规则
{
    int doors;
    int dogs;
    doors = 5;
    dogs = 3;
    // 其他语句
}
```

C99 和 C11 遵循 C++的惯例，可以把声明放在块中的任何位置。尽管如此，首次使用变量之前一定要先声明它。因此，如果编译器支持这一新特性，可以这样编写上面的代码：

```
int main()            // 目前的 C 规则
{
    // 一些语句
    int doors;
    doors = 5; // 第 1 次使用 doors
    // 其他语句
    int dogs;
    dogs = 3; // 第 1 次使用 dogs
    // 其他语句
}
```

为了与旧系统更好地兼容，本书沿用最初的规则（即，把变量声明都写在块的顶部）。

现在，读者可能有 3 个问题：什么是数据类型？如何命名？为何要声明变量？请往下看。

数据类型

C 语言可以处理多种类型的数据，如整数、字符和浮点数。把变量声明为整型或字符类型，计算机才能正确地储存、读取和解释数据。下一章将详细介绍 C 语言中的各种数据类型。

命名

给变量命名时要使用有意义的变量名或标识符（如，程序中需要一个变量数羊，该变量名应该是 sheep_count 而不是 x3）。如果变量名无法清楚地表达自身的用途，可在注释中进一步说明。这是一种良好的编程习惯和编程技巧。

C99 和 C11 允许使用更长的标识符名，但是编译器只识别前 63 个字符。对于外部标识符（参阅第 12 章），只允许使用 31 个字符。〔以前 C90 只允许 6 个字符，这是一个很大的进步。旧式编译器通常最多只允许使用 8 个字符。〕实际上，你可以使用更长的字符，但是编译器会忽略超出的字符。也就是说，如果有两个标识符名都有 63 个字符，只有一个字符不同，那么编译器会识别这是两个不同的名称。如果两个标识符都是 64 个字符，只有最后一个字符不同，那么编译器可能将其视为同一个名称，也可能不会。标准并未定义在这种情况下会发生什么。

可以用小写字母、大写字母、数字和下划线（_）来命名。而且，名称的第 1 个字符必须是字符或下划线，不能是数字。表 2.1 给出了一些示例。

表 2.1 有效和无效的名称

有效的名称	无效的名称
wiggles	$Z]**
cat2	2cat
Hot_Tub	Hot-Tub
taxRate	tax rate
_kcab	don't

操作系统和 C 库经常使用以一个或两个下划线字符开始的标识符（如，_kcab），因此最好避免在自己的程序中使用这种名称。标准标签都以一个或两个下划线字符开始，如库标识符。这样的标识符都是保留的。这意味着，虽然使用它们没有语法错误，但是会导致名称冲突。

C 语言的名称区分大小写，即把一个字母的大写和小写视为两个不同的字符。因此，stars 和 Stars、STARS 都不同。

为了让 C 语言更加国际化，C99 和 C11 根据通用字符名（即 UCN）机制添加了扩展字符集。其中包含了除英文字母以外的部分字符。欲了解详细内容，请参阅附录 B 的“参考资料 VII：扩展字符支持”。

声明变量的 4 个理由

一些更老的语言（如，FORTRAN 和 BASIC 的最初形式）都允许直接使用变量，不必先声明。为何 C 语言不采用这种简单易行的方法？原因如下。

- 把所有的变量放在一处，方便读者查找和理解程序的用途。如果变量名都是有意义的（如，taxtate 而不是 r），这样做效果很好。如果变量名无法表述清楚，在注释中解释变量的含义。这种方法让程序的可读性更高。
- 声明变量会促使你在编写程序之前做一些计划。程序在开始时要获得哪些信息？希望程序如何输出？表示数据最好的方式是什么？
- 声明变量有助于发现隐藏在程序中的小错误，如变量名拼写错误。例如，假设在某些不需要声明就可以直接使用变量的语言中，编写如下语句：

  ```
  RADIUS1 = 20.4;
  ```

  在后面的程序中，误写成：

  ```
  CIRCUM = 6.28 * RADIUSl;
  ```

你不小心把数字 1 打成小写字母 l。这些语言会创建一个新的变量 RADIUSl，并使用该变量中的值（也许是 0，也许是垃圾值），导致赋给 CIRCUM 的值是错误值。你可能要花很久时间才能查出原因。这样的错误在 C 语言中不会发生（除非你很不明智地声明了两个极其相似的变量），因为编译器在发现未声明的 RADIUSl 时会报错。

- 如果事先未声明变量，C 程序将无法通过编译。如果前几个理由还不足以说服你，这个理由总可以让你认真考虑一下了。

如果要声明变量，应该声明在何处？前面提到过，C99 之前的标准要求把声明都置于块的顶部，这样规定的好处是：把声明放在一起更容易理解程序的用途。C99 允许在需要时才声明变量，这样做的好处是：在给变量赋值之前声明变量，就不会忘记给变量赋值。但是实际上，许多编译器都还不支持 C99。

6. 赋值

```
num = 1;
```

程序清单中的这行代码是*赋值表达式语句*[1]。赋值是 C 语言的基本操作之一。该行代码的意思是“把值 1 赋给变量 num”。在执行 int num;声明时，编译器在计算机内存中为变量 num 预留了空间，然后在执行这行赋值表达式语句时，把值储存在之前预留的位置。可以给 num 赋不同的值，这就是 num 之所以被称为变量（*variable*）的原因。注意，该赋值表达式语句从右侧把值赋到左侧。另外，该语句以分号结尾，如图 2.2 所示。



图 2.2　赋值是 C 语言中的基本操作之一

7. printf()函数

```
printf("I am a simple ");
printf("computer.\n");
printf("My favorite number is %d because it is first.\n", num);
```

这 3 行都使用了 C 语言的一个标准函数：printf()。圆括号表明 printf 是一个函数名。圆括号中的内容是从 main()函数传递给 printf()函数的信息。例如，上面的第 1 行把 I am a simple 传递给 printf()函数。该信息被称为参数，或者更确切地说，是函数的实际参数（*actual argument*），如图 2.3 所示。〔在 C 语言中，实际参数（简称实参）是传递给函数的特定值，形式参数（简称形参）是函数中用于储存值的变量。第 5 章中将详述相关内容。〕printf()函数用参数来做什么？该函数会查看双引号中的内容，并将其打印在屏幕上。

[1] C 语言是通过赋值运算符而不是赋值语句完成赋值操作。根据 C 标准，C 语言并没有所谓的“赋值语句”，本书及一些其他书籍中提到的“赋值语句”实际上是表达式语句（C 语言的 6 种基本语句之一）。本书把“赋值语句”均译为“赋值表达式语句”，以提醒初学者注意。——译者注

```
printf("That's mere contrariness");
```



图 2.3 带实参的 printf()函数

第 1 行 `printf()`演示了在 C 语言中如何**调用**函数。只需输入函数名，把所需的参数填入圆括号即可。当程序运行到这一行时，控制权被转给已命名的函数（该例中是 `printf()`）。函数执行结束后，控制权被返回至**主调函数**（*calling function*），该例中是 `main()`。

第 2 行 `printf()`函数的双引号中的`\n` 字符并未输出。这是为什么？`\n` 的意思是换行。`\n` 组合（依次输入这两个字符）代表一个**换行符**（*newline character*）。对于 `printf()`而言，它的意思是“在下一行的最左边开始新的一行”。也就是说，打印换行符的效果与在键盘按下 Enter 键相同。既然如此，为何不在键入 `printf()`参数时直接使用 Enter 键？因为编辑器可能认为这是直接的命令，而不是储存在在源代码中的指令。换句话说，如果直接按下 Enter 键，编辑器会退出当前行并开始新的一行。但是，换行符仅会影响程序输出的显示格式。

换行符是一个**转义序列**（*escape sequence*）。转义序列用于代表难以表示或无法输入的字符。如，`\t` 代表 Tab 键，`\b` 代表 Backspace 键（退格键）。每个转义序列都以反斜杠字符（\）开始。我们在第 3 章中再来探讨相关内容。

这样，就解释了为什么 3 行 `printf()`语句只打印出两行：第 1 个 `printf()`打印的内容中不含换行符，但是第 2 和第 3 个 `printf()`中都有换行符。

第 3 个 `printf()`还有一些不明之处：参数中的`%d` 在打印时有什么作用？先来看该函数的输出：

```
My favorite number is 1 because it is first.
```

对比发现，参数中的`%d` 被数字 `1` 代替了，而 `1` 就是变量 num 的值。`%d` 相当于是一个占位符，其作用是指明输出 num 值的位置。该行和下面的 BASIC 语句很像：

```
PRINT "My favorite number is "; num; " because it is first."
```

实际上，C 语言的 `printf()`比 BASIC 的这条语句做的事情多一些。`%`提醒程序，要在该处打印一个变量，d 表明把变量作为十进制整数打印。`printf()`函数名中的 `f` 提醒用户，这是一种**格式化打印函数**。`printf()`函数有多种打印变量的格式，包括小数和十六进制整数。后面章节在介绍数据类型时，会详细介绍相关内容。

8. return 语句

```
return 0;
```

`return` 语句[1]是程序清单 2.1 的最后一条语句。`int main(void)`中的 `int` 表明 `main()`函数应返回一个整数。C 标准要求 `main()`这样做。有返回值的 C 函数要有 `return` 语句。该语句以 `return` 关键字开始，后面是待返回的值，并以分号结尾。如果遗漏 `main()`函数中的 `return` 语句，程序在运行至最外面的右花括号（`}`）时会返回 `0`。因此，可以省略 `main()`函数末尾的 `return` 语句。但是，不要在其他有返回值的函数中漏掉它。因此，强烈建议读者养成在 `main()`函数中保留 `return` 语句的好习惯。在这种情况下，可将其看作是统一代码风格。但对于某些操作系统（包括 Linux 和 UNIX），`return` 语句有实际的用途。第 11 章再详述这个主题。

[1] 在 C 语言中，return 语句是一种跳转语句。——译者注

2.3　简单程序的结构

在看过一个具体的程序示例后，我们来了解一下 C 程序的基本结构。程序由一个或多个函数组成，必须有 main()函数。函数由函数头和函数体组成。函数头包括函数名、传入该函数的信息类型和函数的返回类型。通过函数名后的圆括号可识别出函数，圆括号里可能为空，可能有参数。函数体被花括号括起来，由一系列语句、声明组成，如图 2.4 所示。本章的程序示例中有一条声明，声明了程序使用的变量名和类型。然后是一条赋值表达式语句，变量被赋给一个值。接下来是 3 条 printf()语句[1]，调用 printf()函数 3 次。最后，main()以 return 语句结束。

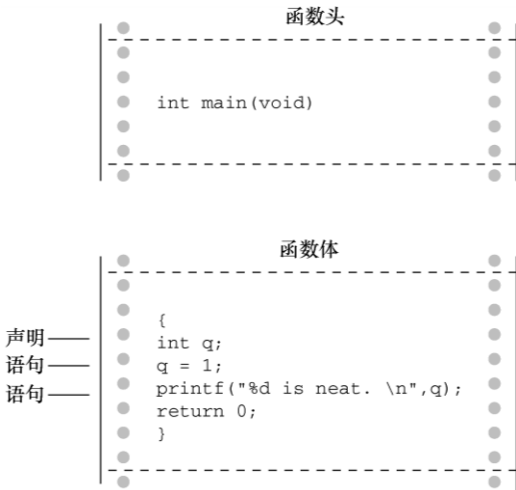


图 2.4　函数包含函数头和函数体

简而言之，一个简单的 C 程序的格式如下：

```
#include <stdio.h>
int main(void)
{
    语句
    return 0;
}
```

（大部分语句都以分号结尾。）

2.4　提高程序可读性的技巧

编写可读性高的程序是良好的编程习惯。可读性高的程序更容易理解，以后也更容易修改和更正。提高程序的可读性还有助于你理清编程思路。

前面介绍过两种提高程序可读性的技巧：选择有意义的函数名和写注释。注意，使用这两种技巧时应相得益彰，避免重复啰嗦。如果变量名是 width，就不必写注释说明该变量表示宽度，但是如果变量名是

[1] 市面上许多书籍（包括本书）都把这种语句叫作“函数调用语句”，但是历年的 C 标准中从来没有函数调用语句！值得一提的是，函数调用本身是一个表达式，圆括号是运算符，圆括号左边的函数名是运算对象。在 C11 标准中，这样的表达式是一种后缀表达式。在表达式末尾加上分号，就成了表达式语句。请初学者注意，这样的“函数调用语句”实质是表达式语句。本书的错误之处已在翻译过程中更正。——译者注

video_routine_4，就要解释一下该变量名的含义。

提高程序可读性的第 3 个技巧是：在函数中用空行分隔概念上的多个部分。例如，程序清单 2.1 中用空行把声明部分和程序的其他部分区分开来。C 语言并未规定一定要使用空行，但是多使用空行能提高程序的可读性。

提高程序可读性的第 4 个技巧是：每条语句各占一行。同样，这也不是 C 语言的要求。C 语言的格式比较自由，可以把多条语句放在一行，也可以每条语句独占一行。下面的语句都没问题，但是不好看：

```
int main( void ) { int four; four
=
4
;
printf(
       "%d\n",
four); return 0;}
```

分号告诉编译器一条语句在哪里结束、下一条语句在哪里开始。如果按照本章示例的约定来编写代码（见图 2.5），程序的逻辑会更清晰。

```
int main(void) /* 把2音寻（测水深的单位）转换成英尺*/——写注释

{
int feet, fathoms;  ——————————— 使用有意义的变量名
                    ——————————— 使用空行
fathoms=2;
feet=6*fathoms;  ——————————— 每行一条语句
printf("There are %d feet in %d fathoms!\n", feet, fathoms);
return 0;
}
```

图 2.5 提高程序的可读性

2.5 进一步使用 C

本章的第 1 个程序相当简单，下面的程序清单 2.2 也不太难。

程序清单 2.2 fathm_ft.c **程序**

```
// fathm_ft.c -- 把 2 音寻转换成英寸
#include <stdio.h>
int main(void)
{
    int feet, fathoms;

    fathoms = 2;
    feet = 6 * fathoms;
    printf("There are %d feet in %d fathoms!\n", feet, fathoms);
    printf("Yes, I said %d feet!\n", 6 * fathoms);

    return 0;
}
```

与程序清单 2.1 相比，以上代码有什么新内容？这段代码提供了程序描述，声明了多个变量，进行了乘法运算，并打印了两个变量的值。下面我们更详细地分析这些内容。

2.5.1　程序说明

程序在开始处有一条注释（使用新的注释风格），给出了文件名和程序的目的。写这种程序说明很简单、不费时，而且在以后浏览或打印程序时很有帮助。

2.5.2　多条声明

接下来，程序在一条声明中声明了两个变量，而不是一个变量。为此，要在声明中用逗号隔开两个变量（feet 和 fathoms）。也就是说，

```
int feet, fathoms;
```

和

```
int feet;
int fathoms;
```

等价。

2.5.3　乘法

然后，程序进行了乘法运算。利用计算机强大的计算能力来计算 6 乘以 2。C 语言和许多其他语言一样，用*表示乘法。因此，语句

```
feet = 6 * fathoms;
```

的意思是“查找变量 fathoms 的值，用 6 乘以该值，并把计算结果赋给变量 feet”。

2.5.4　打印多个值

最后，程序以新的方式使用 printf()函数。如果编译并运行该程序，输出应该是这样：

```
There are 12 feet in 2 fathoms!
Yes, I said 12 feet!
```

程序的第 1 个 printf()中进行了两次替换。双引号号后面的第 1 个变量（feet）替换了双引号中的第 1 个%d；双引号号后面的第 2 个变量（fathoms）替换了双引号中的第 2 个%d。注意，待输出的变量列于双引号的后面。还要注意，变量之间要用逗号隔开。

第 2 个 printf()函数说明待打印的值不一定是变量，只要可求值得出合适类型值的项即可，如 6 * fathoms。

该程序涉及的范围有限，但它是把音寻[1]转换成英寸程序的核心部分。我们还需要把其他值通过交互的方式赋给 feet，其方法将在后面章节中介绍。

2.6　多个函数

到目前为止，介绍的几个程序都只使用了 printf()函数。程序清单 2.3 演示了除 main()以外，如何把自己的函数加入程序中。

---

[1] 音寻，也称为寻。航海用的深度单位，1 英寻=6 英尺=1.8 米，通常用在海图上测量水深。——译者注

程序清单 2.3 `two_func.c` **程序**

```
//* two_func.c -- 一个文件中包含两个函数 */
#include <stdio.h>
void butler(void); /* ANSI/ISO C 函数原型 */
int main(void)
{
    printf("I will summon the butler function.\n");
    butler();
    printf("Yes. Bring me some tea and writeable DVDs.\n");

    return 0;
}
void butler(void) /* 函数定义开始 */
{
    printf("You rang, sir?\n");
}
```

该程序的输出如下：

```
I will summon the butler function.
You rang, sir?
Yes. Bring me some tea and writeable DVDs.
```

`butler()`函数在程序中出现了 3 次。第 1 次是函数原型（*prototype*），告知编译器在程序中要使用该函数；第 2 次以函数调用（*function call*）的形式出现在 `main()`中；最后一次出现在函数定义（*function definition*）中，函数定义即是函数本身的源代码。下面逐一分析。

C90 标准新增了函数原型，旧式的编译器可能无法识别（稍后我们将介绍，如果使用这种编译器应该怎么做）。函数原型是一种声明形式，告知编译器正在使用某函数，因此函数原型也被称为函数声明（*function declaration*）。函数原型还指明了函数的属性。例如，`butler()`函数原型中的第 1 个 `void` 表明，`butler()`函数没有返回值（通常，被调函数会向主调函数返回一个值，但是 `bulter()`函数没有）。第 2 个 `void`（`butler(void)`中的 `void`）的意思是 `butler()`函数不带参数。因此，当编译器运行至此，会检查 `butler()`是否使用得当。注意，`void` 在这里的意思是“空的”，而不是“无效”。

早期的 C 语言支持一种更简单的函数声明，只需指定返回类型，不用描述参数：

```
void butler();
```

早期的 C 代码中的函数声明就类似上面这样，不是现在的函数原型。C90、C99 和 C11 标准都承认旧版本的形式，但是也表明了会逐渐淘汰这种过时的写法。如果要使用以前写的 C 代码，就需要把旧式声明转换成函数原型。本书在后面的章节会继续介绍函数原型的相关内容。

接下来我们继续分析程序。在 `main()`中调用 `butler()`很简单，写出函数名和圆括号即可。当 `butler()`执行完毕后，程序会继续执行 `main()`中的下一条语句。

程序的最后部分是 `butler()`函数的定义，其形式和 `main()`相同，都包含函数头和用花括号括起来的函数体。函数头重述了函数原型的信息：`bulter()`不带任何参数，且没有返回值。如果使用老式编译器，请去掉圆括号中的 `void`。

这里要注意，何时执行 `butler()`函数取决于它在 `main()`中被调用的位置，而不是 `butler()`的定义在文件中的位置。例如，把 `butler()`函数的定义放在 `main()`定义之前，不会改变程序的执行顺序，`butler()`函数仍然在两次 `printf()`调用之间被调用。记住，无论 `main()`在程序文件处于什么位置，所有的 C 程序都从 `main()`开始执行。但是，C 的惯例是把 `main()`放在开头，因为它提供了程序的基本框架。

C 标准建议，要为程序中用到的所有函数提供函数原型。标准 include 文件（包含文件）为标准库函数提供可函数原型。例如，在 C 标准中，stdio.h 文件包含了 printf()的函数原型。第 6 章最后一个示例演示了如何使用带返回值的函数，第 9 章将详细全面地介绍函数。

2.7　调试程序

现在，你可以编写一个简单的 C 程序，但是可能会犯一些简单的错误。程序的错误通常叫做 bug，找出并修正错误的过程叫做调试（*debug*）。程序清单 2.4 是一个有错误的程序，看看你能找出几处。

程序清单 2.4　nogood.c 程序

```
/* nogood.c -- 有错误的程序 */
#include <stdio.h>
int main(void)
(
    int n, int n2, int n3;

    /* 该程序有多处错误
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3)

    return 0;
)
```

2.7.1　语法错误

程序清单 2.4 中有多处语法错误。如果不遵循 C 语言的规则就会犯语法错误。这类似于英文中的语法错误。例如，看看这个句子：*Bugs frustrate be can*[1]。该句子中的英文单词都是有效的单词（即，拼写正确），但是并未按照正确的顺序组织句子，而且用词也不妥。C 语言的语法错误指的是，把有效的 C 符号放在错误的地方。

nogood.c 程序中有哪些错误？其一，main()函数体使用圆括号来代替花括号。这就是把 C 符号用错了地方。其二，变量声明应该这样写：

```
int n, n2, n3;
```

或者，这样写：

```
int n;
int n2;
int n3;
```

其三，main()中的注释末尾漏掉了*/（另一种修改方案是，用//替换/*）。最后，printf()语句末尾漏掉了分号。

如何发现程序的语法错误？首先，在编译之前，浏览源代码看是否能发现一些明显的错误。接下来，查看编译器是否发现错误，检查程序的语法错误是它的工作之一。在编译程序时，编译器发现错误会报告错误信息，指出每一处错误的性质和具体位置。

[1] 要理解该句子存在语法错误，需要具备基本的英文语法知识。——译者注

尽管如此，编译器也有出错的时候。也许某处隐藏的语法错误会导致编译器误判。例如，由于 nogood.c 程序未正确声明 n2 和 n3，会导致编译器在使用这些变量时发现更多问题。实际上，有时不用把编译器报告的所有错误逐一修正，仅修正第 1 条或前几处错误后，错误信息就会少很多。继续这样做，直到编译器不再报错。编译器另一个常见的毛病是，报错的位置比真正的错误位置滞后一行。例如，编译器在编译下一行时才会发现上一行缺少分号。因此，如果编译器报错某行缺少分号，请检查上一行。

2.7.2 语义错误

语义错误是指意思上的错误。例如，考虑这个句子：*Scornful derivatives sing greenly*（轻蔑的衍生物不熟练地唱歌）。句中的形容词、名词、动词和副词都在正确的位置上，所以语法正确。但是，却让人不知所云。在 C 语言中，如果遵循了 C 规则，但是结果不正确，那就是犯了语义错误。程序示例中有这样的错误：

```
n3 = n2 * n2;
```

此处，n3 原意表示 n 的 3 次方，但是代码中的 n3 被设置成 n 的 4 次方（n2 = n * n）。

编译器无法检测语义错误，因为这类错误并未违反 C 语言的规则。编译器无法了解你的真正意图，所以你只能自己找出这些错误。例如，假设你修正了程序的语法错误，程序应该如程序清单 2.5 所示：

程序清单 2.5 stillbad.c **程序**

```
/* stillbad.c -- 修复了语法错误的程序 */
#include <stdio.h>
int main(void)
{
    int n, n2, n3;

    /* 该程序有一个语义错误 */
    n = 5;
    n2 = n * n;
    n3 = n2 * n2;
    printf("n = %d, n squared = %d, n cubed = %d\n", n, n2, n3);

    return 0;
}
```

该程序的输出如下：

```
n = 5, n squared = 25, n cubed = 625
```

如果对简单的立方比较熟悉，就会注意到 625 不对。下一步是跟踪程序的执行步骤，找出程序如何得出这个答案。对于本例，通过查看代码就会发现其中的错误，但是，还应该学习更系统的方法。方法之一是，把自己想象成计算机，跟着程序的步骤一步一步地执行。下面，我们来试试这种方法。

main()函数体一开始就声明了 3 个变量：n、n2、n3。你可以画出 3 个盒子并把变量名写在盒子上来模拟这种情况（见图 2.6）。接下来，程序把 5 赋给变量 n。你可以在标签为 n 的盒子里写上 5。接着，程序把 n 和 n 相乘，并把乘积赋给 n2。因此，查看标签为 n 的盒子，其值是 5，5 乘以 5 得 25，于是把 25 放进标签为 n2 的盒子里。为了模拟下一条语句（n3 = n2 * n2），查看 n2 盒子，发现其值是 25。25 乘以 25 得 625，把 625 放进标签为 n3 的盒子。原来如此！程序中计算的是 n2 的平方，不是用 n2 乘以 n 得到 n 的 3 次方。

对于上面的程序示例，检查程序的过程可能过于繁琐。但是，用这种方法一步一步查看程序的执行情况，通常是发现程序问题所在的良方。

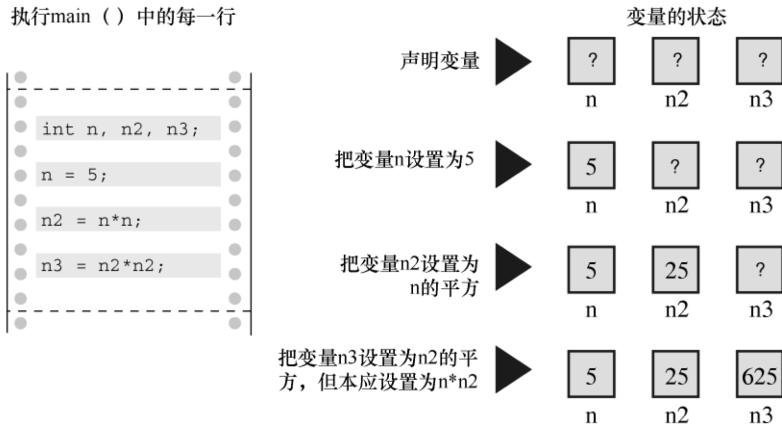


图 2.6　跟踪程序的执行步骤

2.7.3　程序状态

通过逐步跟踪程序的执行步骤，并记录每个变量，便可监视程序的状态。程序状态（*program state*）是在程序的执行过程中，某给定点上所有变量值的集合。它是计算机当前状态的一个快照。

我们刚刚讨论了一种跟踪程序状态的方法：自己模拟计算机逐步执行程序。但是，如果程序中有 10000 次循环，这种方法恐怕行不通。不过，你可以跟踪一小部分循环，看看程序是否按照预期的方式执行。另外，还要考虑一种情况：你很可能按照自己所想去执行程序，而不是根据实际写出来的代码去执行。因此，要尽量忠实代码来模拟。

定位语义错误的另一种方法是：在程序中的关键点插入额外的 `printf()`语句，以监视制定变量值的变化。通过查看值的变化可以了解程序的执行情况。对程序的执行满意后，便可删除额外的 `printf()`语句，然后重新编译。

检测程序状态的第 3 种方法是使用调试器。调试器（*debugger*）是一种程序，让你一步一步运行另一个程序，并检查该程序变量的值。调试器有不同的使用难度和复杂度。较高级的调试器会显示正在执行的源代码行号。这在检查有多条执行路径的程序时很方便，因为很容易知道正在执行哪条路径。如果你的编译器自带调试器，现在可以花点时间学会怎么使用它。例如，试着调试一下程序清单 2.4。

2.8　关键字和保留标识符

关键字是 C 语言的词汇。它们对 C 而言比较特殊，不能用它们作为标识符（如，变量名）。许多关键字用于指定不同的类型，如 `int`。还有一些关键字（如，`if`）用于控制程序中语句的执行顺序。在表 2.2 中所列的 C 语言关键字中，粗体表示的是 C90 标准新增的关键字，斜体表示的 C99 标准新增的关键字，粗斜体表示的是 C11 标准新增的关键字。

表 2.2　ISO C 关键字

auto	extern	short	while
break	float	**signed**	***_Alignas***
case	for	sizeof	***_Alignof***
char	goto	static	***_Atomic***

续表

const	if	struct	***_Bool***
continue	*inline*	switch	***_Complex***
default	int	typedef	***_Generic***
do	long	union	***_Imaginary***
double	register	unsigned	***_Noreturn***
else	restrict	void	***_Static_assert***
enum	return	**volatile**	***_Thread_local***

如果使用关键字不当（如，用关键字作为变量名），编译器会将其视为语法错误。还有一些保留标识符（*reserved identifier*），C 语言已经指定了它们的用途或保留它们的使用权，如果你使用这些标识符来表示其他意思会导致一些问题。因此，尽管它们也是有效的名称，不会引起语法错误，也不能随便使用。保留标识符包括那些以下划线字符开头的标识符和标准库函数名，如 printf()。

2.9　关键概念

编程是一件富有挑战性的事情。程序员要具备抽象和逻辑的思维，并谨慎地处理细节问题（编译器会强迫你注意细节问题）。平时和朋友交流时，可能用错几个单词，犯一两个语法错误，或者说几句不完整的句子，但是对方能明白你想说什么。而编译器不允许这样，对它而言，几乎正确仍然是错误。

编译器不会在下面讲到的概念性问题上帮助你。因此，本书在这一章中介绍一些关键概念帮助读者弥补这部分的内容。

在本章中，读者的目标应该是理解什么是 C 程序。可以把程序看作是你希望计算机如何完成任务的描述。编译器负责处理一些细节工作，例如把你要计算机完成的任务转换成底层的机器语言（如果从量化方面来解释编译器所做的工作，它可以把 1KB 的源文件创建成 60KB 的可执行文件；即使是一个很简单的 C 程序也要用大量的机器语言来表示）。由于编译器不具有真正的智能，所以你必须用编译器能理解的术语表达你的意图，这些术语就是 C 语言标准规定的形式规则（尽管有些约束，但总比直接用机器语言方便得多）。

编译器希望接收到特定格式的指令，我们在本章已经介绍过。作为程序员的任务是，在符合 C 标准的编译器框架中，表达你希望程序应该如何完成任务的想法。

2.10　本章小结

C 程序由一个或多个 C 函数组成。每个 C 程序必须包含一个 main()函数，这是 C 程序要调用的第 1 个函数。简单的函数由函数头和后面的一对花括号组成，花括号中是由声明、语句组成的函数体。

在 C 语言中，大部分语句都以分号结尾。声明为变量创建变量名和标识该变量中储存的数据类型。变量名是一种标识符。赋值表达式语句把值赋给变量，或者更一般地说，把值赋给存储空间。函数表达式语句用于调用指定的已命名函数。调用函数执行完毕后，程序会返回到函数调用后面的语句继续执行。

printf()函数用于输出想要表达的内容和变量的值。

一门语言的*语法*是一套规则，用于管理语言中各有效语句组合在一起的方式。语句的*语义*是语句要表达的意思。编译器可以检测出语法错误，但是程序里的语义错误只有在编译完之后才能从程序的行为中表现出来。检查程序是否有语义错误要跟踪程序的状态，即程序每执行一步后所有变量的值。

最后，关键字是 C 语言的词汇。

2.11　复习题

复习题的参考答案在附录 A 中。

1．C 语言的基本模块是什么？

2．什么是语法错误？写出一个英语例子和 C 语言例子。

3．什么是语义错误？写出一个英语例子和 C 语言例子。

4．Indiana Sloth 编写了下面的程序，并征求你的意见。请帮助他评定。

```
include studio.h
int main{void} /* 该程序打印一年有多少周 /*
(
    int s

    s := 56;
    print(There are s weeks in a year.);
    return 0;
```

5．假设下面的 4 个例子都是完整程序中的一部分，它们都输出什么结果？

```
a. printf("Baa Baa Black Sheep.");
   printf("Have you any wool?\n");
b. printf("Begone!\nO creature of lard!\n");
c. printf("What?\nNo/nfish?\n");
d. int num;
   num = 2;
   printf("%d + %d = %d", num, num, num + num);
```

6．在 main、int、function、char、=中，哪些是 C 语言的关键字？

7．如何以下面的格式输出变量 words 和 lines 的值（这里，3020 和 350 代表两个变量的值）？

```
There were 3020 words and 350 lines.
```

8．考虑下面的程序：

```
#include <stdio.h>
int main(void)
{
    int a, b;

    a = 5;
    b = 2; /* 第 7 行 */
    b = a; /* 第 8 行 */
    a = b; /* 第 9 行 */
    printf("%d %d\n", b, a);
    return 0;
}
```

请问，在执行完第 7、第 8、第 9 行后，程序的状态分别是什么？

9．考虑下面的程序：

```
#include <stdio.h>
int main(void)
{
    int x, y;

    x = 10;
```

```
    y = 5;      /* 第 7 行 */
    y = x + y;  /*第 8 行*/
    x = x*y;    /*第 9 行*/
    printf("%d %d\n", x, y);
    return 0;
}
```

请问，在执行完第 7、第 8、第 9 行后，程序的状态分别是什么？

2.12 编程练习

纸上得来终觉浅，绝知此事要躬行。读者应该试着编写一两个简单的程序，体会一下编写程序是否和阅读本章介绍的这样轻松。题目中会给出一些建议，但是应该尽量自己思考这些问题。一些编程答案练习的答案可在出版商网站获取。

1. 编写一个程序，调用一次 printf()函数，把你的姓名打印在一行。再调用一次 printf()函数，把你的姓名分别打印在两行。然后，再调用两次 printf()函数，把你的姓名打印在一行。输出应如下所示（当然要把示例的内容换成你的姓名）：

```
Gustav Mahler        ←第 1 次打印的内容
Gustav               ←第 2 次打印的内容
Mahler               ←仍是第 2 次打印的内容
Gustav Mahler        ←第 3 次和第 4 次打印的内容
```

2. 编写一个程序，打印你的姓名和地址。
3. 编写一个程序把你的年龄转换成天数，并显示这两个值。这里不用考虑闰年的问题。
4. 编写一个程序，生成以下输出：

```
For he's a jolly good fellow!
For he's a jolly good fellow!
For he's a jolly good fellow!
Which nobody can deny!
```

除了 `main()`函数以外，该程序还要调用两个自定义函数：一个名为 `jolly()`，用于打印前 3 条消息，调用一次打印一条；另一个函数名为 `deny()`，打印最后一条消息。

5. 编写一个程序，生成以下输出：

```
Brazil, Russia, India, China
India, China,
Brazil, Russia
```

除了 `main()`以外，该程序还要调用两个自定义函数：一个名为 `br()`，调用一次打印一次“`Brazil, Russia`”；另一个名为 `ic()`，调用一次打印一次“`India, China`”。其他内容在 `main()`函数中完成。

6. 编写一个程序，创建一个整型变量 `toes`，并将 `toes` 设置为 `10`。程序中还要计算 `toes` 的两倍和 `toes` 的平方。该程序应打印 3 个值，并分别描述以示区分。
7. 许多研究表明，微笑益处多多。编写一个程序，生成以下格式的输出：

```
Smile!Smile!Smile!
Smile!Smile!
Smile!
```

该程序要定义一个函数，该函数被调用一次打印一次“`Smile!`”，根据程序的需要使用该函数。

8. 在 C 语言中，函数可以调用另一个函数。编写一个程序，调用一个名为 `one_three()`的函数。该

函数在一行打印单词“one”，再调用第 2 个函数 two()，然后在另一行打印单词“three”。two()函数在一行显示单词“two”。main()函数在调用 one_three()函数前要打印短语“starting now:”，并在调用完毕后显示短语“done!”。因此，该程序的输出应如下所示：

```
starting now:
one
two
three
done!
```

第 3 章

数据和 C

本章介绍以下内容：

- 关键字：int 、short、long、unsigned、char、float、double、_Bool、_Complex、_Imaginary
- 运算符：sizeof()
- 函数：scanf()
- 整数类型和浮点数类型的区别
- 如何书写整型和浮点型常数，如何声明这些类型的变量
- 如何使用 printf()和 scanf()函数读写不同类型的值

程序离不开数据。把数字、字母和文字输入计算机，就是希望它利用这些数据完成某些任务。例如，需要计算一份利息或显示一份葡萄酒商的排序列表。本章除了介绍如何读取数据外，还将教会读者如何操控数据。

C 语言提供两大系列的多种数据类型。本章详细介绍两大数据类型：整数类型和浮点数类型，讲解这些数据类型是什么、如何声明它们、如何以及何时使用它们。除此之外，还将介绍常量和变量的区别。读者很快就能看到第 1 个交互式程序。

3.1 示例程序

本章仍从一个简单的程序开始。如果发现有不熟悉的内容，别担心，我们稍后会详细解释。该程序的意图比较明了，请试着编译并运行程序清单 3.1 中的源代码。为了节省时间，在输入源代码时可省略注释。

程序清单 3.1 platinum.c **程序**

```
/* platinum.c  -- your weight in platinum */
#include <stdio.h>
int main(void)
{
    float weight;    /* 你的体重                 */
    float value;     /* 相等重量的白金价值       */

    printf("Are you worth your weight in platinum?\n");
    printf("Let's check it out.\n");
    printf("Please enter your weight in pounds: ");

    /* 获取用户的输入                            */
    scanf("%f", &weight);
    /* 假设白金的价格是每盎司$1700           */
```

```
    /* 14.5833 用于把英镑常衡盎司转换为金衡盎司1*/
    value = 1700.0 * weight * 14.5833;
    printf("Your weight in platinum is worth $%.2f.\n", value);
    printf("You are easily worth that! If platinum prices drop,\n");
    printf("eat more to maintain your value.\n");

    return 0;
}
```

提示　错误与警告

如果输入程序时打错（如，漏了一个分号），编译器会报告语法错误消息。然而，即使输入正确无误，编译器也可能给出一些警告，如“警告：从 double 类型转换成 float 类型可能会丢失数据”。错误消息表明程序中有错，不能进行编译。而警告则表明，尽管编写的代码有效，但可能不是程序员想要的。警告并不终止编译。特殊的警告与 C 如何处理 1700.0 这样的值有关。本例不必理会这个问题，本章稍后会进一步说明。

输入该程序时，可以把 1700.0 改成贵金属白金当前的市价，但是不要改动 14.5833，该数是 1 英镑的金衡盎司数（金衡盎司用于衡量贵金属，而英镑常衡盎司用于衡量人的体重）。

注意，“enter your weight”的意思是输入你的体重，然后按下 Enter 或 Return 键（不要键入体重后就一直等着）。按下 Enter 键是告知计算机，你已完成输入数据。该程序需要你输入一个数字（如，155），而不是单词（如，too much）。如果输入字母而不是数字，会导致程序出问题。这个问题要用 if 语句来解决（详见第 7 章），因此请先输入数字。下面是程序的输出示例：

```
Are you worth your weight in platinum?
Let's check it out.
Please enter your weight in pounds: 156
Your weight in platinum is worth $3867491.25.
You are easily worth that! If platinum prices drop,
eat more to maintain your value.
```

程序调整

即使用第 2 章介绍的方法，在程序中添加下面一行代码：

```
getchar();
```

程序的输出是否依旧在屏幕上一闪而过？本例，需要调用两次 getchar()函数：

```
getchar();
getchar();
```

getchar()函数读取下一个输入字符，因此程序会等待用户输入。在这种情况下，键入 156 并按下 Enter（或 Return）键（发送一个换行符），然后 scanf()读取键入的数字，第 1 个 getchar()读取换行符，第 2 个 getchar()让程序暂停，等待输入。

[1] 欧美日常使用的度量衡单位是常衡盎司（*avoirdupois ounce*），而欧美黄金市场上使用的黄金交易计量单位是金衡盎司（*troy ounce*）。国际黄金市场上的报价，其单位“盎司”都指的是黄金盎司。常衡盎司属英制计量单位，做重量单位时也称为英两。相关换算参考如下：1 常衡盎司 = 28.350 克，1 金衡盎司 = 31.104 克，16 常衡盎司 = 1 磅。该程序的单位转换思路是：把磅换算成金衡盎司，即 28.350÷31.104×16=14.5833。——译者注

3.1.1　程序中的新元素

程序清单 3.1 中包含 C 语言的一些新元素。

- 注意，代码中使用了一种新的变量声明。前面的例子中只使用了整数类型的变量（int），但是本例使用了浮点数类型（float）的变量，以便处理更大范围的数据。float 类型可以储存带小数的数字。
- 程序中演示了常量的几种新写法。现在可以使用带小数点的数了。
- 为了打印新类型的变量，在 printf()中使用%f 来处理浮点值。%.2f 中的.2 用于精确控制输出，指定输出的浮点数只显示小数点后面两位。
- scanf()函数用于读取键盘的输入。%f 说明 scanf()要读取用户从键盘输入的浮点数，&weight 告诉 scanf()把输入的值赋给名为 weight 的变量。scanf()函数使用&符号表明找到 weight 变量的地点。下一章将详细讨论&。就目前而言，请按照这样写。
- 也许本程序最突出的新特点是它的交互性。计算机向用户询问信息，然后用户输入数字。与非交互式程序相比，交互式程序用起来更有趣。更重要的是，交互式使得程序更加灵活。例如，示例程序可以使用任何合理的体重，而不只是 156 磅。不必重写程序，就可以根据不同体重进行计算。scanf()和 printf()函数用于实现这种交互。scanf()函数读取用户从键盘输入的数据，并把数据传递给程序；printf()函数读取程序中的数据，并把数据显示在屏幕上。把两个函数结合起来，就可以建立人机双向通信（见图 3.1），这让使用计算机更加饶有趣味。

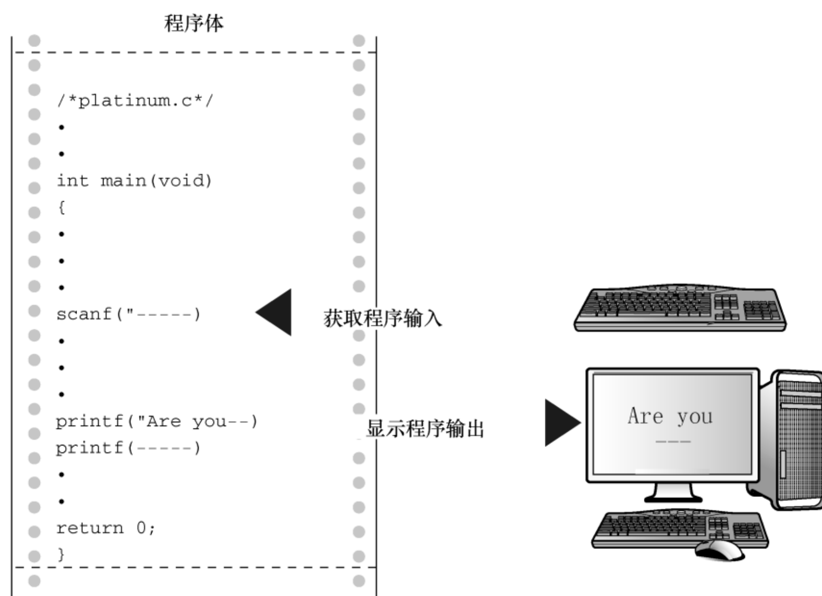


图 3.1　程序中的 scanf()和 printf()函数

本章着重解释上述新特性中的前两项：各种数据类型的变量和常量。第 4 章将介绍后 3 项。

3.2　变量与常量数据

在程序的指导下，计算机可以做许多事情，如数值计算、名字排序、执行语言或视频命令、计算彗星轨道、准备邮件列表、拨电话号码、画画、做决策或其他你能想到的事情。要完成这些任务，程序需要使用数据，即承载信息的数字和字符。有些数据类型在程序使用之前已经预先设定好了，在整个程序的运行过程中没有变化，这些称为常量（*constant*）。其他数据类型在程序运行期间可能会改变或被赋值，这些称为变量（*variable*）。在示例程序中，`weight` 是一个变量，`14.5833` 是一个常量。那么，`1700.0` 是常量还是变量？在现实生活中，白金的价格不会是常量，但是在程序中，像 `1700.0` 这样的价格被视为常量。

3.3　数据：数据类型关键字

不仅变量和常量不同，不同的数据类型之间也有差异。一些数据类型表示数字，一些数据类型表示字母（更普遍地说是字符）。C 通过识别一些基本的数据类型来区分和使用这些不同的数据类型。如果数据是常量，编译器一般通过用户书写的形式来识别类型（如，42 是整数，42.100 是浮点数）。但是，对变量而言，要在声明时指定其类型。稍后会详细介绍如何声明变量。现在，我们先来了解一下 C 语言的基本类型关键字。K&C 给出了 7 个与类型相关的关键字。C90 标准添加了 2 个关键字，C99 标准又添加了 3 个关键字（见表 3.1）。

表 3.1　C 语言的数据类型关键字

最初 K&R 给出的关键字	C90 标准添加的关键字	C99 标准添加的关键字
int	signed	_Bool
long	void	_Complex
short		_Imaginary
unsigned		
char		
float		
double		

在 C 语言中，用 `int` 关键字来表示基本的整数类型。后 3 个关键字（`long`、`short` 和 `unsigned`）和 C90 新增的 `signed` 用于提供基本整数类型的变式，例如 `unsigned short int` 和 `long long int`。`char` 关键字用于指定字母和其他字符（如，`#`、`$`、`%`和`*`）。另外，`char` 类型也可以表示较小的整数。`float`、`double` 和 `long double` 表示带小数点的数。`_Bool` 类型表示布尔值（`true` 或 `false`），`_complex` 和 `_Imaginary` 分别表示复数和虚数。

通过这些关键字创建的类型，按计算机的储存方式可分为两大基本类型：整数类型和浮点数类型。

位、字节和字

位、字节和字是描述计算机数据单元或存储单元的术语。这里主要指存储单元。

最小的存储单元是位（*bit*），可以储存 0 或 1（或者说，位用于设置“开”或“关”）。虽然 1 位储存的信息有限，但是计算机中位的数量十分庞大。位是计算机内存的基本构建块。

字节（*byte*）是常用的计算机存储单位。对于几乎所有的机器，1 字节均为 8 位。这是字节的标准定义，至少在衡量存储单位时是这样（但是，C 语言对此有不同的定义，请参阅本章 3.4.3 节）。既然 1 位可以表示 0 或 1，那么 8 位字节就有 256（2 的 8 次方）种可能的 0、1 的组合。通过二进制编码（仅

用 0 和 1 便可表示数字），便可表示 0～255 的整数或一组字符（第 15 章将详细讨论二进制编码，如果感兴趣可以现在浏览一下该章的内容）。

字（*word*）是设计计算机时给定的自然存储单位。对于 8 位的微型计算机（如，最初的苹果机），1 个字长只有 8 位。从那以后，个人计算机字长增至 16 位、32 位，直到目前的 64 位。计算机的字长越大，其数据转移越快，允许的内存访问也更多。

3.3.1　整数和浮点数

整数类型？浮点数类型？如果觉得这些术语非常陌生，别担心，下面先简述它们的含义。如果不熟悉位、字节和字的概念，请阅读上面方框中的内容。刚开始学习时，不必了解所有的细节，就像学习开车之前不必详细了解汽车内部引擎的原理一样。但是，了解一些计算机或汽车引擎内部的原理会对你有所帮助。

对我们而言，整数和浮点数的区别是它们的书写方式不同。对计算机而言，它们的区别是储存方式不同。下面详细介绍整数和浮点数。

3.3.2　整数

和数学的概念一样，在 C 语言中，整数是没有小数部分的数。例如，2、−23 和 2456 都是整数。而 3.14、0.22 和 2.000 都不是整数。计算机以二进制数字储存整数，例如，整数 7 以二进制写是 111。因此，要在 8 位字节中储存该数字，需要把前 5 位都设置成 0，后 3 位设置成 1（如图 3.2 所示）。

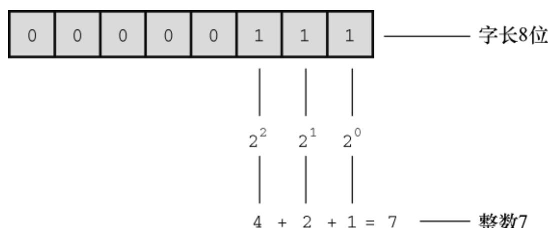


图 3.2　使用二进制编码储存整数 7

3.3.3　浮点数

浮点数与数学中实数的概念差不多。2.75、3.16E7、7.00 和 2e-8 都是浮点数。注意，在一个值后面加上一个小数点，该值就成为一个浮点值。所以，7 是整数，7.00 是浮点数。显然，书写浮点数有多种形式。稍后将详细介绍 e 记数法，这里先做简要介绍：3.16E7 表示 3.16×10^7（3.16 乘以 10 的 7 次方）。其中，10^7=10000000，7 被称为 10 的指数。

这里关键要理解浮点数和整数的储存方案不同。计算机把浮点数分成小数部分和指数部分来表示，而且分开储存这两部分。因此，虽然 7.00 和 7 在数值上相同，但是它们的储存方式不同。在十进制下，可以把 7.0 写成 0.7E1。这里，0.7 是小数部分，1 是指数部分。图 3.3 演示了一个储存浮点数的例子。当然，计算机在内部使用二进制和 2 的幂进行储存，而不是 10 的幂。第 15 章将详述相关内容。现在，我们着重讲解这两种类型的实际区别。

- 整数没有小数部分，浮点数有小数部分。
- 浮点数可以表示的范围比整数大。参见本章末的表 3.3。
- 对于一些算术运算（如，两个很大的数相减），浮点数损失的精度更多。

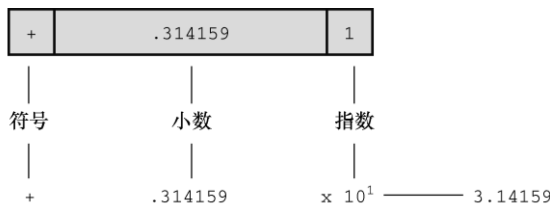


图 3.3　以浮点格式（十进制）储存 π 的值

- 因为在任何区间内（如，1.0 到 2.0 之间）都存在无穷多个实数，所以计算机的浮点数不能表示区间内所有的值。浮点数通常只是实际值的近似值。例如，7.0 可能被储存为浮点值 6.99999。稍后会讨论更多精度方面的内容。
- 过去，浮点运算比整数运算慢。不过，现在许多 CPU 都包含浮点处理器，缩小了速度上的差距。

3.4　C 语言基本数据类型

本节将详细节介绍 C 语言的基本数据类型，包括如何声明变量、如何表示字面值常量（如，5 或 2.78），以及典型的用法。一些老式的 C 语言编译器无法支持这里提到的所有类型，请查阅你使用的编译器文档，了解可以使用哪些类型。

3.4.1　int 类型

C 语言提供了许多整数类型，为什么一种类型不够用？因为 C 语言让程序员针对不同情况选择不同的类型。特别是，C 语言中的整数类型可表示不同的取值范围和正负值。一般情况使用 int 类型即可，但是为满足特定任务和机器的要求，还可以选择其他类型。

int 类型是有符号整型，即 int 类型的值必须是整数，可以是正整数、负整数或零。其取值范围依计算机系统而异。一般而言，储存一个 int 要占用一个机器字长。因此，早期的 16 位 IBM PC 兼容机使用 16 位来储存一个 int 值，其取值范围（即 int 值的取值范围）是-32768～32767。目前的个人计算机一般是 32 位，因此用 32 位储存一个 int 值。现在，个人计算机产业正逐步向着 64 位处理器发展，自然能储存更大的整数。ISO C 规定 int 的取值范围最小为-32768～32767。一般而言，系统用一个特殊位的值表示有符号整数的正负号。第 15 章将介绍常用的方法。

1. 声明 int 变量

第 2 章中已经用 int 声明过基本整型变量。先写上 int，然后写变量名，最后加上一个分号。要声明多个变量，可以单独声明每个变量，也可在 int 后面列出多个变量名，变量名之间用逗号分隔。下面都是有效的声明：

```
int erns;
int hogs, cows, goats;
```

可以分别在 4 条声明中声明各变量，也可以在一条声明中声明 4 个变量。两种方法的效果相同，都为 4 个 int 大小的变量赋予名称并分配内存空间。

以上声明创建了变量，但是并没有给它们提供值。变量如何获得值？前面介绍过在程序中获取值的两种途径。第 1 种途径是赋值：

```
cows = 112;
```

第 2 种途径是，通过函数（如，scanf()）获得值。接下来，我们着重介绍第 3 种途径。

2. 初始化变量

初始化（*initialize*）变量就是为变量赋一个初始值。在 C 语言中，初始化可以直接在声明中完成。只需在变量名后面加上赋值运算符（=）和待赋给变量的值即可。如下所示：

```
int hogs = 21;
int cows = 32, goats = 14;
int dogs, cats = 94; /* 有效，但是这种格式很糟糕 */
```

以上示例的最后一行，只初始化了 cats，并未初始化 dogs。这种写法很容易让人误认为 dogs 也被初始化为 94，所以最好不要把初始化的变量和未初始化的变量放在同一条声明中。

简而言之，声明为变量创建和标记存储空间，并为其指定初始值（如图 3.4 所示）。

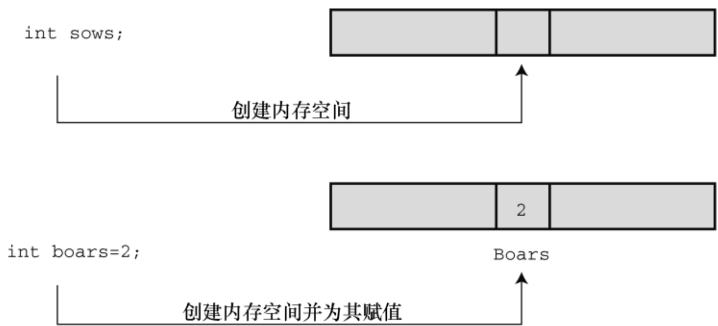


图 3.4　定义并初始化变量

3. int 类型常量

上面示例中出现的整数（21、32、14 和 94）都是*整型常量*或*整型字面量*。C 语言把不含小数点和指数的数作为整数。因此，22 和-44 都是整型常量，但是 22.0 和 2.2E1 则不是。C 语言把大多数整型常量视为 int 类型，但是非常大的整数除外。详见后面“long 常量和 long long 常量”小节对 long int 类型的讨论。

4. 打印 int 值

可以使用 printf()函数打印 int 类型的值。第 2 章中介绍过，%d 指明了在一行中打印整数的位置。%d 称为*转换说明*，它指定了 printf()应使用什么格式来显示一个值。格式化字符串中的每个%d 都与待打印变量列表中相应的 int 值匹配。这个值可以是 int 类型的变量、int 类型的常量或其他任何值为 int 类型的表达式。作为程序员，要确保转换说明的数量与待打印值的数量相同，编译器不会捕获这类型的错误。程序清单 3.2 演示了一个简单的程序，程序中初始化了一个变量，并打印该变量的值、一个常量值和一个简单表达式的值。另外，程序还演示了如果粗心犯错会导致什么结果。

程序清单 3.2　print1.c 程序

```
/* print1.c - 演示printf()的一些特性 */
#include <stdio.h>
int main(void)
{
    int ten = 10;
    int two = 2;

    printf("Doing it right: ");
    printf("%d minus %d is %d\n", ten, 2, ten - two);
    printf("Doing it wrong: ");
    printf("%d minus %d is %d\n", ten);  // 遗漏2个参数
```

```
    return 0;
}
```

编译并运行该程序，输出如下：

```
Doing it right: 10 minus 2 is 8
Doing it wrong: 10 minus 16 is 1650287143
```

在第一行输出中，第 1 个%d 对应 int 类型变量 ten；第 2 个%d 对应 int 类型常量 2；第 3 个%d 对应 int 类型表达式 ten - two 的值。在第二行输出中，第 1 个%d 对应 ten 的值，但是由于没有给后两个%d 提供任何值，所以打印出的值是内存中的任意值（读者在运行该程序时显示的这两个数值会与输出示例中的数值不同，因为内存中储存的数据不同，而且编译器管理内存的位置也不同）。

你可能会抱怨编译器为何不能捕获这种明显的错误，但实际上问题出在 printf()不寻常的设计。大部分函数都需要指定数目的参数，编译器会检查参数的数目是否正确。但是，printf()函数的参数数目不定，可以有 1 个、2 个、3 个或更多，编译器也爱莫能助。记住，使用 printf()函数时，要确保转换说明的数量与待打印值的数量相等。

5. 八进制和十六进制

通常，C 语言都假定整型常量是十进制数。然而，许多程序员很喜欢使用八进制和十六进制数。因为 8 和 16 都是 2 的幂，而 10 却不是。显然，八进制和十六进制记数系统在表达与计算机相关的值时很方便。例如，十进制数 65536 经常出现在 16 位机中，用十六进制表示正好是 10000。另外，十六进制数的每一位的数恰好由 4 位二进制数表示。例如，十六进制数 3 是 0011，十六进制数 5 是 0101。因此，十六进制数 35 的位组合（*bit pattern*）是 00110101，十六进制数 53 的位组合是 01010011。这种对应关系使得十六进制和二进制的转换非常方便。但是，计算机如何知道 10000 是十进制、十六进制还是二进制？在 C 语言中，用特定的前缀表示使用哪种进制。0x 或 0X 前缀表示十六进制值，所以十进制数 16 表示成十六进制是 0x10 或 0X10。与此类似，0 前缀表示八进制。例如，十进制数 16 表示成八进制是 020。第 15 章将更全面地介绍进制相关的内容。

要清楚，使用不同的进制数是为了方便，不会影响数被储存的方式。也就是说，无论把数字写成 16、020 或 0x10，储存该数的方式都相同，因为计算机内部都以二进制进行编码。

6. 显示八进制和十六进制

在 C 程序中，既可以使用和显示不同进制的数。不同的进制要使用不同的转换说明。以十进制显示数字，使用%d；以八进制显示数字，使用%o；以十六进制显示数字，使用%x。另外，要显示各进制数的前缀 0、0x 和 0X，必须分别使用%#o、%#x、%#X。程序清单 3.3 演示了一个小程序。回忆一下，在某些集成开发环境（IDE）下编写的代码中插入 getchar();语句，程序在执行完毕后不会立即关闭执行窗口。

程序清单 3.3　bases.c 程序

```
/* bases.c--以十进制、八进制、十六进制打印十进制数 100 */
#include <stdio.h>
int main(void)
{
    int x = 100;

    printf("dec = %d; octal = %o; hex = %x\n", x, x, x);
    printf("dec = %d; octal = %#o; hex = %#x\n", x, x, x);

    return 0;
}
```

编译并运行该程序，输出如下：

```
dec = 100; octal = 144; hex = 64
dec = 100; octal = 0144; hex = 0x64
```

该程序以3种不同记数系统显示同一个值。printf()函数做了相应的转换。注意，如果要在八进制和十六进制值前显示0和0x前缀，要分别在转换说明中加入#。

3.4.2 其他整数类型

初学C语言时，int类型应该能满足大多数程序的整数类型需求。尽管如此，还应了解一下整型的其他形式。当然，也可以略过本节跳至3.4.3节阅读char类型的相关内容，以后有需要时再阅读本节。

C语言提供3个附属关键字修饰基本整数类型：short、long和unsigned。应记住以下几点。

- short int类型（或者简写为short）占用的存储空间可能比int类型少，常用于较小数值的场合以节省空间。与int类似，short是有符号类型。
- long int或long占用的存储空间可能比int多，适用于较大数值的场合。与int类似，long是有符号类型。
- long long int或long long（C99标准加入）占用的储存空间可能比long多，适用于更大数值的场合。该类型至少占64位。与int类似，long long是有符号类型。
- unsigned int或unsigned只用于非负值的场合。这种类型与有符号类型表示的范围不同。例如，16位unsigned int允许的取值范围是0～65535，而不是-32768～32767。用于表示正负号的位现在用于表示另一个二进制位，所以无符号整型可以表示更大的数。
- 在C90标准中，添加了unsigned long int或unsigned long和unsigned int或unsigned short类型。C99标准又添加了unsigned long long int或unsigned long long。
- 在任何有符号类型前面添加关键字signed，可强调使用有符号类型的意图。例如，short、short int、signed short、signed short int都表示同一种类型。

1. 声明其他整数类型

其他整数类型的声明方式与int类型相同，下面列出了一些例子。不是所有的C编译器都能识别最后3条声明，最后一个例子所有的类型是C99标准新增的。

```
long int estine;
long johns;
short int erns;
short ribs;
unsigned int s_count;
unsigned players;
unsigned long headcount;
unsigned short yesvotes;
long long ago;
```

2. 使用多种整数类型的原因

为什么说short类型“可能”比int类型占用的空间少，long类型“可能”比int类型占用的空间多？因为C语言只规定了short占用的存储空间不能多于int，long占用的存储空间不能少于int。这样规定是为了适应不同的机器。例如，过去的一台运行Windows 3的机器上，int类型和short类型都占16位，long类型占32位。后来，Windows和苹果系统都使用16位储存short类型，32位储存int类型和long类型（使用32位可以表示的整数数值超过20亿）。现在，计算机普遍使用64位处理器，为了

储存 64 位的整数，才引入了 long long 类型。

现在，个人计算机上最常见的设置是，long long 占 64 位，long 占 32 位，short 占 16 位，int 占 16 位或 32 位（依计算机的自然字长而定）。原则上，这 4 种类型代表 4 种不同的大小，但是在实际使用中，有些类型之间通常有重叠。

C 标准对基本数据类型只规定了允许的最小大小。对于 16 位机，short 和 int 的最小取值范围是[−32767,32767]；对于 32 位机，long 的最小取值范围是[−2147483647,2147483647]。对于 unsigned short 和 unsigned int，最小取值范围是[0,65535]；对于 unsigned long，最小取值范围是[0,4294967295]。long long 类型是为了支持 64 位的需求，最小取值范围是[−9223372036854775807,9223372036854775807]；unsigned long long 的最小取值范围是[0,18446744073709551615]。如果要开支票，这个数是一千八百亿亿（兆）六千七百四十四万亿零七百三十七亿零九百五十五万一千六百一十五。但是，谁会去数？

int 类型那么多，应该如何选择？首先，考虑 unsigned 类型。这种类型的数常用于计数，因为计数不用负数。而且，unsigned 类型可以表示更大的正数。

如果一个数超出了 int 类型的取值范围，且在 long 类型的取值范围内时，使用 long 类型。然而，对于那些 long 占用的空间比 int 大的系统，使用 long 类型会减慢运算速度。因此，如非必要，请不要使用 long 类型。另外要注意一点：如果在 long 类型和 int 类型占用空间相同的机器上编写代码，当确实需要 32 位的整数时，应使用 long 类型而不是 int 类型，以便把程序移植到 16 位机后仍然可以正常工作。类似地，如果确实需要 64 位的整数，应使用 long long 类型。

如果在 int 设置为 32 位的系统中要使用 16 位的值，应使用 short 类型以节省存储空间。通常，只有当程序使用相对于系统可用内存较大的整型数组时，才需要重点考虑节省空间的问题。使用 short 类型的另一个原因是，计算机中某些组件使用的硬件寄存器是 16 位。

3. long 常量和 long long 常量

通常，程序代码中使用的数字（如，2345）都被储存为 int 类型。如果使用 1000000 这样的大数字，超出了 int 类型能表示的范围，编译器会将其视为 long int 类型（假设这种类型可以表示该数字）。如果数字超出 long 可表示的最大值，编译器则将其视为 unsigned long 类型。如果还不够大，编译器则将其视为 long long 或 unsigned long long 类型（前提是编译器能识别这些类型）。

八进制和十六进制常量被视为 int 类型。如果值太大，编译器会尝试使用 unsigned int。如果还不够大，编译器会依次使用 long、unsigned long、long long 和 unsigned long long 类型。

有些情况下，需要编译器以 long 类型储存一个小数字。例如，编程时要显式使用 IBM PC 上的内存地址时。另外，一些 C 标准函数也要求使用 long 类型的值。要把一个较小的常量作为 long 类型对待，可以在值的末尾加上 l（小写的 L）或 L 后缀。使用 L 后缀更好，因为 l 看上去和数字 1 很像。因此，在 int 为 16 位、long 为 32 位的系统中，会把 7 作为 16 位储存，把 7L 作为 32 位储存。l 或 L 后缀也可用于八进制和十六进制整数，如 020L 和 0x10L。

类似地，在支持 long long 类型的系统中，也可以使用 ll 或 LL 后缀来表示 long long 类型的值，如 3LL。另外，u 或 U 后缀表示 unsigned long long，如 5ull、10LLU、6LLU 或 9Ull。

整数溢出

如果整数超出了相应类型的取值范围会怎样？下面分别将有符号类型和无符号类型的整数设置为比最大值略大，看看会发生什么（printf()函数使用%u 说明显示 unsigned int 类型的值）。

```
/* toobig.c-- 超出系统允许的最大 int 值*/
```

```
#include <stdio.h>
int main(void)
{
    int i = 2147483647;
     unsigned int j = 4294967295;

     printf("%d %d %d\n", i, i+1, i+2);
     printf("%u %u %u\n", j, j+1, j+2);

     return 0;
}
```

在我们的系统下输出的结果是：

```
2147483647    -2147483648  -2147483647
4294967295    0    1
```

可以把无符号整数 j 看作是汽车的里程表。当达到它能表示的最大值时，会重新从起始点开始。整数 i 也是类似的情况。它们主要的区别是，在超过最大值时，unsigned int 类型的变量 j 从 0 开始；而 int 类型的变量 i 则从−2147483648 开始。注意，当 i 超出（溢出）其相应类型所能表示的最大值时，系统并未通知用户。因此，在编程时必须自己注意这类问题。

溢出行为是未定义的行为，C 标准并未定义有符号类型的溢出规则。以上描述的溢出行为比较有代表性，但是也可能会出现其他情况。

4. 打印 short、long、long long 和 unsigned 类型

打印 unsigned int 类型的值，使用%u 转换说明；打印 long 类型的值，使用%ld 转换说明。如果系统中 int 和 long 的大小相同，使用%d 就行。但是，这样的程序被移植到其他系统（int 和 long 类型的大小不同）中会无法正常工作。在 x 和 o 前面可以使用 l 前缀，%lx 表示以十六进制格式打印 long 类型整数，%lo 表示以八进制格式打印 long 类型整数。注意，虽然 C 允许使用大写或小写的常量后缀，但是在转换说明中只能用小写。

C 语言有多种 printf()格式。对于 short 类型，可以使用 h 前缀。%hd 表示以十进制显示 short 类型的整数，%ho 表示以八进制显示 short 类型的整数。h 和 l 前缀都可以和 u 一起使用，用于表示无符号类型。例如，%lu 表示打印 unsigned long 类型的值。程序清单 3.4 演示了一些例子。对于支持 long long 类型的系统，%lld 和%llu 分别表示有符号和无符号类型。第 4 章将详细介绍转换说明。

程序清单 3.4　print2.c 程序

```
/* print2.c--更多 printf()的特性 */
#include <stdio.h>
int main(void)
{
     unsigned int un = 3000000000; /* int 为 32 位和 short 为 16 位的系统 */
     short end = 200;
     long big = 65537;
     long long verybig = 12345678908642;

     printf("un = %u and not %d\n", un, un);
     printf("end = %hd and %d\n", end, end);
     printf("big = %ld and not %hd\n", big, big);
     printf("verybig= %lld and not %ld\n", verybig, verybig);

     return 0;
}
```

在特定的系统中输出如下（输出的结果可能不同）：

```
un = 3000000000 and not -1294967296
end = 200 and 200
big = 65537 and not 1
verybig= 12345678908642 and not 1942899938
```

该例表明，使用错误的转换说明会得到意想不到的结果。第 1 行输出，对于无符号变量 un，使用%d 会生成负值！其原因是，无符号值 3000000000 和有符号值−129496296 在系统内存中的内部表示完全相同（详见第 15 章）。因此，如果告诉 printf()该数是无符号数，它打印一个值；如果告诉它该数是有符号数，它将打印另一个值。在待打印的值大于有符号值的最大值时，会发生这种情况。对于较小的正数（如 96），有符号和无符号类型的存储、显示都相同。

第 2 行输出，对于 short 类型的变量 end，在 printf()中无论指定以 short 类型（%hd）还是 int 类型（%d）打印，打印出来的值都相同。这是因为在给函数传递参数时，C 编译器把 short 类型的值自动转换成 int 类型的值。你可能会提出疑问：为什么要进行转换？h 修饰符有什么用？第 1 个问题的答案是，int 类型被认为是计算机处理整数类型时最高效的类型。因此，在 short 和 int 类型的大小不同的计算机中，用 int 类型的参数传递速度更快。第 2 个问题的答案是，使用 h 修饰符可以显示较大整数被截断成 short 类型值的情况。第 3 行输出就演示了这种情况。把 65537 以二进制格式写成一个 32 位数是 00000000000000010000000000000001。使用%hd，printf()只会查看后 16 位，所以显示的值是 1。与此类似，输出的最后一行先显示了 verybig 的完整值，然后由于使用了%ld，printf()只显示了储存在后 32 位的值。

本章前面介绍过，程序员必须确保转换说明的数量和待打印值的数量相同。以上内容也提醒读者，程序员还必须根据待打印值的类型使用正确的转换说明。

> **提示　匹配 printf()说明符的类型**
>
> 在使用 printf()函数时，切记检查每个待打印值都有对应的转换说明，还要检查转换说明的类型是否与待打印值的类型相匹配。

3.4.3　使用字符：char 类型

char 类型用于储存字符（如，字母或标点符号），但是从技术层面看，char 是整数类型。因为 char 类型实际上储存的是整数而不是字符。计算机使用数字编码来处理字符，即用特定的整数表示特定的字符。美国最常用的编码是 ASCII 编码，本书也使用此编码。例如，在 ASCII 码中，整数 65 代表大写字母 A。因此，储存字母 A 实际上储存的是整数 65（许多 IBM 的大型主机使用另一种编码——EBCDIC，其原理相同。另外，其他国家的计算机系统可能使用完全不同的编码）。

标准 ASCII 码的范围是 0～127，只需 7 位二进制数即可表示。通常，char 类型被定义为 8 位的存储单元，因此容纳标准 ASCII 码绰绰有余。许多其他系统（如 IMB PC 和苹果 Macs）还提供扩展 ASCII 码，也在 8 位的表示范围之内。一般而言，C 语言会保证 char 类型足够大，以储存系统（实现 C 语言的系统）的基本字符集。

许多字符集都超过了 127，甚至多于 255。例如，日本汉字（*kanji*）字符集。商用的统一码（*Unicode*）创建了一个能表示世界范围内多种字符集的系统，目前包含的字符已超过 110000 个。国际标准化组织（ISO）和国际电工技术委员会（IEC）为字符集开发了 ISO/IEC 10646 标准。统一码标准也与 ISO/IEC 10646 标准兼容。

C 语言把 1 字节定义为 char 类型占用的位（*bit*）数，因此无论是 16 位还是 32 位系统，都可以使用 char 类型。

1. 声明 char 类型变量

char 类型变量的声明方式与其他类型变量的声明方式相同。下面是一些例子：

```
char response;
char itable, latan;
```

以上声明创建了 3 个 char 类型的变量：response、itable 和 latan。

2. 字符常量和初始化

如果要把一个字符常量初始化为字母 A，不必背下 ASCII 码，用计算机语言很容易做到。通过以下初始化把字母 A 赋给 grade 即可：

```
char grade = 'A';
```

在 C 语言中，用单引号括起来的单个字符被称为**字符常量**（*character constant*）。编译器一发现'A'，就会将其转换成相应的代码值。单引号必不可少。下面还有一些其他的例子：

```
char broiled;        /* 声明一个 char 类型的变量 */
broiled = 'T';       /* 为其赋值，正确 */
broiled = T;         /* 错误！此时 T 是一个变量 */
broiled = "T";       /* 错误！此时"T"是一个字符串 */
```

如上所示，如果省略单引号，编译器认为 T 是一个变量名；如果把 T 用双引号括起来，编译器则认为"T"是一个字符串。字符串的内容将在第 4 章中介绍。

实际上，字符是以数值形式储存的，所以也可使用数字代码值来赋值：

```
char grade = 65; /* 对于 ASCII，这样做没问题，但这是一种不好的编程风格 */
```

在本例中，虽然 65 是 int 类型，但是它在 char 类型能表示的范围内，所以将其赋值给 grade 没问题。由于 65 是字母 A 对应的 ASCII 码，因此本例是把 A 赋给 grade。注意，能这样做的前提是系统使用 ASCII 码。其实，用'A'代替 65 才是较为妥当的做法，这样在任何系统中都不会出问题。因此，最好使用字符常量，而不是数字代码值。

奇怪的是，C 语言将字符常量视为 int 类型而非 char 类型。例如，在 int 为 32 位、char 为 8 位的 ASCII 系统中，有下面的代码：

```
char grade = 'B';
```

本来'B'对应的数值 66 储存在 32 位的存储单元中，现在却可以储存在 8 位的存储单元中（grade）。利用字符常量的这种特性，可以定义一个字符常量'FATE'，即把 4 个独立的 8 位 ASCII 码储存在一个 32 位存储单元中。如果把这样的字符常量赋给 char 类型变量 grade，只有最后 8 位有效。因此，grade 的值是'E'。

3. 非打印字符

单引号只适用于字符、数字和标点符号，浏览 ASCII 表会发现，有些 ASCII 字符打印不出来。例如，一些代表行为的字符（如，退格、换行、终端响铃或蜂鸣）。C 语言提供了 3 种方法表示这些字符。

第 1 种方法前面介绍过——使用 ASCII 码。例如，蜂鸣字符的 ASCII 值是 7，因此可以这样写：

```
char beep = 7;
```

第 2 种方法是，使用特殊的符号序列表示一些特殊的字符。这些符号序列叫作**转义序列**（*escape sequence*）。表 3.2 列出了转义序列及其含义。

把转义序列赋给字符变量时，必须用单引号把转义序列括起来。例如，假设有下面一行代码：

```
char nerf = '\n';
```

稍后打印变量 nerf 的效果是，在打印机或屏幕上另起一行。

表 3.2　转义序列

转义序列	含义
\a	警报（ANSI C）
\b	退格
\f	换页
\n	换行
\r	回车
\t	水平制表符
\v	垂直制表符
\\	反斜杠（\）
\'	单引号
\"	双引号
\?	问号
\0oo	八进制值（oo 必须是有效的八进制数，即每个 o 可表示 0～7 中的一个数）
\xhh	十六进制值（hh 必须是有效的十六进制数，即每个 h 可表示 0～f 中的一个数）

现在，我们来仔细分析一下转义序列。使用 C90 新增的警报字符（\a）是否能产生听到或看到的警报，取决于计算机的硬件，蜂鸣是最常见的警报（在一些系统中，警报字符不起作用）。C 标准规定警报字符不得改变活跃位置。标准中的活跃位置（*active position*）指的是显示设备（屏幕、电传打字机、打印机等）中下一个字符将出现的位置。简而言之，平时常说的屏幕光标位置就是活跃位置。在程序中把警报字符输出在屏幕上的效果是，发出一声蜂鸣，但不会移动屏幕光标。

接下来的转义字符\b、\f、\n、\r、\t 和\v 是常用的输出设备控制字符。了解它们最好的方式是查看它们对活跃位置的影响。换页符（\f）把活跃位置移至下一页的开始处；换行符（\n）把活跃位置移至下一行的开始处；回车符（\r）把活跃位置移动到当前行的开始处；水平制表符（\t）将活跃位置移至下一个水平制表点（通常是第 1 个、第 9 个、第 17 个、第 25 个等字符位置）；垂直制表符（\v）把活跃位置移至下一个垂直制表点。

这些转义序列字符不一定在所有的显示设备上都起作用。例如，换页符和垂直制表符在 PC 屏幕上会生成奇怪的符号，光标并不会移动。只有将其输出到打印机上时才会产生前面描述的效果。

接下来的 3 个转义序列（\\、\'、\"）用于打印\、'、"字符（由于这些字符用于定义字符常量，是 printf()函数的一部分，若直接使用它们会造成混乱）。如果打印下面一行内容：

```
Gramps sez, "a \ is a backslash."
```

应这样编写代码：

```
printf("Gramps sez, \"a \\ is a backslash.\"\n");
```

表 3.2 中的最后两个转义序列（\0oo 和\xhh）是 ASCII 码的特殊表示。如果要用八进制 ASCII 码表示一个字符，可以在编码值前面加一个反斜杠（\）并用单引号括起来。例如，如果编译器不识别警报字符（\a），可以使用 ASCII 码来代替：

```
beep = '\007';
```

可以省略前面的 0，'\07'甚至'\7'都可以。即使没有前缀 0，编译器在处理这种写法时，仍会解释为八进制。

从 C90 开始，不仅可以用十进制、八进制形式表示字符常量，C 语言还提供了第 3 种选择——用十六进制形式表示字符常量，即反斜杠后面跟一个 x 或 X，再加上 1～3 位十六进制数字。例如，Ctrl+P 字符的 ASCII 十六进制码是 10（相当于十进制的 16），可表示为'\x10'或'\x010'。图 3.5 列出了一些整数类型的不同进制形式。

整型常量的例子			
类型	十六进制	八进制	十进制
char	\0x41	\0101	N.A.
int	0x41	0101	65
unsigned int	0x41u	0101u	65u
long	0x41L	0101L	65L
unsigned long	0x41UL	0101UL	65UL
long long	0x41LL	0101LL	65LL
unsigned long long	0x41ULL	0101ULL	65ULL

图 3.5　int 系列类型的常量写法示例

使用 ASCII 码时，注意数字和数字字符的区别。例如，字符 4 对应的 ASCII 码是 52。'4'表示字符 4，而不是数值 4。

关于转义序列，读者可能有下面 3 个问题。

- 上面最后一个例子（printf("Gramps sez, \"a \\ is a backslash\"\"n")，为何没有用单引号把转义序列括起来？无论是普通字符还是转义序列，只要是双引号括起来的字符集合，就无需用单引号括起来。双引号中的字符集合叫作字符串（详见第 4 章）。注意，该例中的其他字符（G、r、a、m、p、s 等）都没有用单引号括起来。与此类似，printf("Hello!\007\n");将打印 Hello!并发出一声蜂鸣，而 printf("Hello!7\n");则打印 Hello!7。不是转义序列中的数字将作为普通字符被打印出来。
- 何时使用 ASCII 码？何时使用转义序列？如果要在转义序列（假设使用'\f'）和 ASCII 码（'\014'）之间选择，请选择前者（即'\f'）。这样的写法不仅更好记，而且可移植性更高。'\f'在不使用 ASCII 码的系统中，仍然有效。
- 如果要使用 ASCII 码，为何要写成'\032'而不是 032？首先，'\032'能更清晰地表达程序员使用字符编码的意图。其次，类似\032 这样的转义序列可以嵌入 C 的字符串中，如 printf("Hello!\007\n");中就嵌入了\007。

4. 打印字符

printf()函数用%c 指明待打印的字符。前面介绍过，一个字符变量实际上被储存为 1 字节的整数值。因此，如果用%d 转换说明打印 char 类型变量的值，打印的是一个整数。而%c 转换说明告诉 printf()打印该整数值对应的字符。程序清单 3.5 演示了打印 char 类型变量的两种方式。

程序清单 3.5　charcode.c 程序

```
/* charcode.c-显示字符的代码编号 */
#include <stdio.h>
int main(void)
{
    char ch;

    printf("Please enter a character.\n");
    scanf("%c", &ch);   /* 用户输入字符 */
    printf("The code for %c is %d.\n", ch, ch);

    return 0;
}
```

运行该程序后，输出示例如下：

```
Please enter a character.
C
The code for C is 67.
```

运行该程序时，在输入字母后不要忘记按下 Enter 或 Return 键。随后，scanf()函数会读取用户输入的字符，&符号表示把输入的字符赋给变量 ch。接着，printf()函数打印 ch 的值两次，第 1 次打印一个字符（对应代码中的%c），第 2 次打印一个十进制整数值（对应代码中的%d）。注意，printf()函数中的转换说明决定了数据的显示方式，而不是数据的储存方式（见图 3.6）。

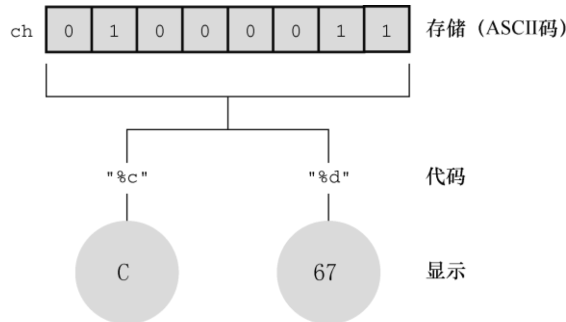


图 3.6　数据显示和数据存储

5. 有符号还是无符号

有些 C 编译器把 char 实现为有符号类型，这意味着 char 可表示的范围是-128～127。而有些 C 编译器把 char 实现为无符号类型，那么 char 可表示的范围是 0～255。请查阅相应的编译器手册，确定正在使用的编译器如何实现 char 类型。或者，可以查阅 limits.h 头文件。下一章将详细介绍头文件的内容。

根据 C90 标准，C 语言允许在关键字 char 前面使用 signed 或 unsigned。这样，无论编译器默认 char 是什么类型，signed char 表示有符号类型，而 unsigned char 表示无符号类型。这在用 char 类型处理小整数时很有用。如果只用 char 处理字符，那么 char 前面无需使用任何修饰符。

3.4.4　_Bool 类型

C99 标准添加了_Bool 类型，用于表示布尔值，即逻辑值 true 和 false。因为 C 语言用值 1 表示 true，值 0 表示 false，所以_Bool 类型实际上也是一种整数类型。但原则上它仅占用 1 位存储空间，因为对 0 和 1 而言，1 位的存储空间足够了。

程序通过布尔值可选择执行哪部分代码。我们将在第 6 章和第 7 章中详述相关内容。

3.4.5 可移植类型：stdint.h 和 inttypes.h

C 语言提供了许多有用的整数类型。但是，某些类型名在不同系统中的功能不一样。C99 新增了两个头文件 stdint.h 和 inttypes.h，以确保 C 语言的类型在各系统中的功能相同。

C 语言为现有类型创建了更多类型名。这些新的类型名定义在 stdint.h 头文件中。例如，int32_t 表示 32 位的有符号整数类型。在使用 32 位 int 的系统中，头文件会把 int32_t 作为 int 的别名。不同的系统也可以定义相同的类型名。例如，int 为 16 位、long 为 32 位的系统会把 int32_t 作为 long 的别名。然后，使用 int32_t 类型编写程序，并包含 stdint.h 头文件时，编译器会把 int 或 long 替换成与当前系统匹配的类型。

上面讨论的类型别名是*精确宽度整数类型*（*exact-width integer type*）的示例。int32_t 表示整数类型的宽度正好是 32 位。但是，计算机的底层系统可能不支持。因此，精确宽度整数类型是可选项。

如果系统不支持精确宽度整数类型怎么办？C99 和 C11 提供了第 2 类别名集合。一些类型名保证所表示的类型一定是至少有指定宽度的最小整数类型。这组类型集合被称为*最小宽度类型*（*minimum width type*）。例如，int_least8_t 是可容纳 8 位有符号整数值的类型中宽度最小的类型的一个别名。如果某系统的最小整数类型是 16 位，可能不会定义 int8_t 类型。尽管如此，该系统仍可使用 int_least8_t 类型，但可能把该类型实现为 16 位的整数类型。

当然，一些程序员更关心速度而非空间。为此，C99 和 C11 定义了一组可使计算达到最快的类型集合。这组类型集合被称为*最快最小宽度类型*（*fastst minimum width type*）。例如，int_fast8_t 被定义为系统中对 8 位有符号值而言运算最快的整数类型的别名。

另外，有些程序员需要系统的最大整数类型。为此，C99 定义了最大的有符号整数类型 intmax_t，可储存任何有效的有符号整数值。类似地，unitmax_t 表示最大的无符号整数类型。顺带一提，这些类型有可能比 long long 和 unsigned long 类型更大，因为 C 编译器除了实现标准规定的类型以外，还可利用 C 语言实现其他类型。例如，一些编译器在标准引入 long long 类型之前，已提前实现了该类型。

C99 和 C11 不仅提供可移植的类型名，还提供相应的输入和输出。例如，printf()打印特定类型时要求与相应的转换说明匹配。如果要打印 int32_t 类型的值，有些定义使用%d，而有些定义使用%ld，怎么办？C 标准针对这一情况，提供了一些字符串宏（第 4 章中详细介绍）来显示可移植类型。例如，inttypes.h 头文件中定义了 PRId32 字符串宏，代表打印 32 位有符号值的合适转换说明（如 d 或 l）。程序清单 3.6 演示了一种可移植类型和相应转换说明的用法。

程序清单 3.6 altnames.c 程序

```
/* altnames.c -- 可移植整数类型名 */
#include <stdio.h>
#include <inttypes.h> // 支持可移植类型
int main(void)
{
    int32_t me32;     // me32 是一个 32 位有符号整型变量

    me32 = 45933945;
    printf("First, assume int32_t is int: ");
    printf("me32 = %d\n", me32);
    printf("Next, let's not make any assumptions.\n");
```

```
    printf("Instead, use a \"macro\" from inttypes.h: ");
    printf("me32 = %" PRId32 "\n", me32);

    return 0;
}
```

该程序最后一个 printf()中，参数 PRId32 被定义在 inttypes.h 中的"d"替换，因而这条语句等价于：

```
printf("me16 = %" "d" "\n", me16);
```

在 C 语言中，可以把多个连续的字符串组合成一个字符串，所以这条语句又等价于：

```
printf("me16 = %d\n", me16);
```

下面是该程序的输出，注意，程序中使用了\"转义序列来显示双引号：

```
First, assume int32_t is int: me32 = 45933945
Next, let's not make any assumptions.
Instead, use a "macro" from inttypes.h: me32 = 45933945
```

篇幅有限，无法介绍扩展的所有整数类型。本节主要是为了让读者知道，在需要时可进行这种级别的类型控制。附录 B 中的参考资料 VI“扩展的整数类型”介绍了完整的 inttypes.h 和 stdint.h 头文件。

> **注意　　对 C99/C11 的支持**
>
> C 语言发展至今，虽然 ISO 已发布了 C11 标准，但是编译器供应商对 C99 的实现程度却各不相同。在本书第 6 版的编写过程中，一些编译器仍未实现 inttypes.h 头文件及其相关功能。

3.4.6　float、double 和 long double

各种整数类型对大多数软件开发项目而言够用了。然而，面向金融和数学的程序经常使用浮点数。C 语言中的浮点类型有 float、double 和 long double 类型。它们与 FORTRAN 和 Pascal 中的 real 类型一致。前面提到过，浮点类型能表示包括小数在内更大范围的数。浮点数的表示类似于科学记数法（即用小数乘以 10 的幂来表示数字）。该记数系统常用于表示非常大或非常小的数。表 3.3 列出了一些示例。

表 3.3　记数法示例

数字	科学记数法	指数记数法
1000000000	1.0×10^{9}	1.0e9
123000	1.23×10^{5}	1.23e5
322.56	3.2256×10^{2}	3.2256e2
0.000056	5.6×10^{-5}	5.6e-5

第 1 列是一般记数法；第 2 列是科学记数法；第 3 列是指数记数法（或称为 e 记数法），这是科学记数法在计算机中的写法，e 后面的数字代表 10 的指数。图 3.7 演示了更多的浮点数写法。

C 标准规定，float 类型必须至少能表示 6 位有效数字，且取值范围至少是 10^{-37}～10^{+37}。前一项规定指 float 类型必须至少精确表示小数点后的 6 位有效数字，如 33.333333。后一项规定用于方便地表示诸如太阳质量（2.0e30 千克）、一个质子的电荷量（1.6e-19 库仑）或国家债务之类的数字。通常，系统储存一个浮点数要占用 32 位。其中 8 位用于表示指数的值和符号，剩下 24 位用于表示非指数部分（也叫作尾数或有效数）及其符号。

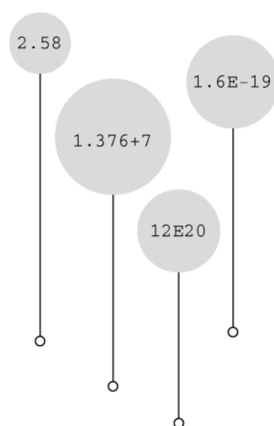


图 3.7 更多浮点数写法示例

C 语言提供的另一种浮点类型是 `double`（意为双精度）。`double` 类型和 `float` 类型的最小取值范围相同，但至少必须能表示 10 位有效数字。一般情况下，`double` 占用 64 位而不是 32 位。一些系统将多出的 32 位全部用来表示非指数部分，这不仅增加了有效数字的位数（即提高了精度），而且还减少了舍入误差。另一些系统把其中的一些位分配给指数部分，以容纳更大的指数，从而增加了可表示数的范围。无论哪种方法，`double` 类型的值至少有 13 位有效数字，超过了标准的最低位数规定。

C 语言的第 3 种浮点类型是 `long double`，以满足比 `double` 类型更高的精度要求。不过，C 只保证 `long double` 类型至少与 `double` 类型的精度相同。

1. 声明浮点型变量

浮点型变量的声明和初始化方式与整型变量相同，下面是一些例子：

```
float noah, jonah;
double trouble;
float planck = 6.63e-34;
long double gnp;
```

2. 浮点型常量

在代码中，可以用多种形式书写浮点型常量。浮点型常量的基本形式是：有符号的数字（包括小数点），后面紧跟 e 或 E，最后是一个有符号数表示 10 的指数。下面是两个有效的浮点型常量：

```
-1.56E+12
2.87e-3
```

正号可以省略。可以没有小数点（如，2E5）或指数部分（如，19.28），但是不能同时省略两者。可以省略小数部分（如，3.E16）或整数部分（如，.45E-6），但是不能同时省略两者。下面是更多的有效浮点型常量示例：

```
3.14159
.2
4e16
.8E-5
100.
```

不要在浮点型常量中间加空格：`1.56 E+12`（错误!）

默认情况下，编译器假定浮点型常量是 `double` 类型的精度。例如，假设 `some` 是 `float` 类型的变量，编写下面的语句：

```
some = 4.0 * 2.0;
```

通常，4.0 和 2.0 被储存为 64 位的 double 类型，使用双精度进行乘法运算，然后将乘积截断成 float 类型的宽度。这样做虽然计算精度更高，但是会减慢程序的运行速度。

在浮点数后面加上 f 或 F 后缀可覆盖默认设置，编译器会将浮点型常量看作 float 类型，如 2.3f 和 9.11E9F。使用 l 或 L 后缀使得数字成为 long double 类型，如 54.3l 和 4.32L。注意，建议使用 L 后缀，因为字母 l 和数字 1 很容易混淆。没有后缀的浮点型常量是 double 类型。

C99 标准添加了一种新的浮点型常量格式——用十六进制表示浮点型常量，即在十六进制数前加上十六进制前缀（0x 或 0X），用 p 和 P 分别代替 e 和 E，用 2 的幂代替 10 的幂（即，p 计数法）。如下所示：

```
0xa.1fp10
```

十六进制 a 等于十进制 10，.1f 是 1/16 加上 15/256（十六进制 f 等于十进制 15），p10 是 2^{10} 或 1024。0xa.1fp10 表示的值是(10 + 1/16 + 15/256)×1024（即，十进制 10364.0）。

注意，并非所有的编译器都支持 C99 的这一特性。

3. 打印浮点值

printf()函数使用%f 转换说明打印十进制记数法的 float 和 double 类型浮点数，用%e 打印指数记数法的浮点数。如果系统支持十六进制格式的浮点数，可用 a 和 A 分别代替 e 和 E。打印 long double 类型要使用%Lf、%Le 或%La 转换说明。给那些未在函数原型中显式说明参数类型的函数（如，printf()）传递参数时，C 编译器会把 float 类型的值自动转换成 double 类型。程序清单 3.7 演示了这些特性。

程序清单 3.7　showf_pt.c 程序

```
/* showf_pt.c -- 以两种方式显示 float 类型的值 */
#include <stdio.h>
int main(void)
{
    float aboat = 32000.0;
    double abet = 2.14e9;
    long double dip = 5.32e-5;

    printf("%f can be written %e\n", aboat, aboat);
    // 下一行要求编译器支持 C99 或其中的相关特性
    printf("And it's %a in hexadecimal, powers of 2 notation\n", aboat);
    printf("%f can be written %e\n", abet, abet);
    printf("%Lf can be written %Le\n", dip, dip);

    return 0;
}
```

该程序的输出如下，前提是编译器支持 C99/C11：

```
32000.000000 can be written 3.200000e+04
And it's 0x1.f4p+14 in hexadecimal, powers of 2 notation
2140000000.000000 can be written 2.140000e+09
0.000053 can be written 5.320000e-05
```

该程序示例演示了默认的输出效果。下一章将介绍如何通过设置字段宽度和小数位数来控制输出格式。

4. 浮点值的上溢和下溢

假设系统的最大 float 类型值是 3.4E38，编写如下代码：

```
float toobig = 3.4E38 * 100.0f;
printf("%e\n", toobig);
```

会发生什么？这是一个上溢（*overflow*）的示例。当计算导致数字过大，超过当前类型能表达的范围时，就会发生上溢。这种行为在过去是未定义的，不过现在 C 语言规定，在这种情况下会给 toobig 赋一个表示无穷大的特定值，而且 printf()显示该值为 inf 或 infinity（或者具有无穷含义的其他内容）。

当除以一个很小的数时，情况更为复杂。回忆一下，float 类型的数以指数和尾数部分来储存。存在这样一个数，它的指数部分是最小值，即由全部可用位表示的最小尾数值。该数字是 float 类型能用全部精度表示的最小数字。现在把它除以 2。通常，这个操作会减小指数部分，但是假设的情况中，指数已经是最小值了。所以计算机只好把尾数部分的位向右移，空出第 1 个二进制位，并丢弃最后一个二进制数。以十进制为例，把一个有 4 位有效数字的数（如，0.1234E-10）除以 10，得到的结果是 0.0123E-10。虽然得到了结果，但是在计算过程中却损失了原末尾有效位上的数字。这种情况叫作下溢（*underflow*）。C 语言把损失了类型全精度的浮点值称为低于正常的（*subnormal*）浮点值。因此，把最小的正浮点数除以 2 将得到一个低于正常的值。如果除以一个非常大的值，会导致所有的位都为 0。现在，C 库已提供了用于检查计算是否会产生低于正常值的函数。

还有另一个特殊的浮点值 NaN（not a number 的缩写）。例如，给 asin()函数传递一个值，该函数将返回一个角度，该角度的正弦就是传入函数的值。但是正弦值不能大于 1，因此，如果传入的参数大于 1，该函数的行为是未定义的。在这种情况下，该函数将返回 NaN 值，printf()函数可将其显示为 nan、NaN 或其他类似的内容。

浮点数舍入错误

给定一个数，加上 1，再减去原来给定的数，结果是多少？你一定认为是 1。但是，下面的浮点运算给出了不同的答案：

```
/* floaterr.c--演示舍入错误 */
#include <stdio.h>
int main(void)
{
    float a,b;

    b = 2.0e20 + 1.0;
    a = b - 2.0e20;
    printf("%f \n", a);

    return 0;
}
```

该程序的输出如下：

```
0.000000        ←Linux 系统下的老式 gcc
-13584010575872.000000 ←Turbo C 1.5
4008175468544.000000   ←XCode 4.5、Visual Studio 2012、当前版本的 gcc
```

得出这些奇怪答案的原因是，计算机缺少足够的小数位来完成正确的运算。2.0e20 是 2 后面有 20 个 0。如果把该数加 1，那么发生变化的是第 21 位。要正确运算，程序至少要储存 21 位数字。而 float 类型的数字通常只能储存按指数比例缩小或放大的 6 或 7 位有效数字。在这种情况下，计算结果一定是错误的。另一方面，如果把 2.0e20 改成 2.0e4，计算结果就没问题。因为 2.0e4 加 1 只需改变第 5 位上的数字，float 类型的精度足够进行这样的计算。

浮点数表示法

上一个方框中列出了由于计算机使用的系统不同，一个程序有不同的输出。原因是，根据前面介

> 绍的知识，实现浮点数表示法的方法有多种。为了尽可能地统一实现，电子和电气工程师协会（IEEE）为浮点数计算和表示法开发了一套标准。现在，许多硬件浮点单元都采用该标准。2011 年，该标准被 ISO/IEC/IEEE 60559:2011 标准收录。该标准作为 C99 和 C11 的可选项，符合硬件要求的平台可开启。`floaterr.c` 程序的第 3 个输出示例即是支持该浮点标准的系统显示的结果。支持 C 标准的编译器还包含捕获异常问题的工具。详见附录 B.5，参考资料 V。

3.4.7　复数和虚数类型

许多科学和工程计算都要用到复数和虚数。C99 标准支持复数类型和虚数类型，但是有所保留。一些独立实现，如嵌入式处理器的实现，就不需要使用复数和虚数（VCR 芯片就不需要复数）。一般而言，虚数类型都是可选项。C11 标准把整个复数软件包都作为可选项。

简而言之，C 语言有 3 种复数类型：`float_Complex`、`double_Complex` 和 `long double _Complex`。例如，`float _Complex` 类型的变量应包含两个 `float` 类型的值，分别表示复数的实部和虚部。类似地，C 语言的 3 种虚数类型是 `float _Imaginary`、`double _Imaginary` 和 `long double _Imaginary`。

如果包含 `complex.h` 头文件，便可用 `complex` 代替`_Complex`，用 `imaginary` 代替`_Imaginary`，还可以用 `I` 代替`-1` 的平方根。

为何 C 标准不直接用 `complex` 作为关键字来代替`_Complex`，而要添加一个头文件（该头文件中把 `complex` 定义为`_Complex`）？因为标准委员会考虑到，如果使用新的关键字，会导致以该关键字作为标识符的现有代码全部失效。例如，之前的 C99，许多程序员已经使用 `struct complex` 定义一个结构来表示复数或者心理学程序中的心理状况（关键字 `struct` 用于定义能储存多个值的结构，详见第 14 章）。让 `complex` 成为关键字会导致之前的这些代码出现语法错误。但是，使用 `struct _Complex` 的人很少，特别是标准使用首字母是下划线的标识符作为预留字以后。因此，标准委员会选定`_Complex` 作为关键字，在不用考虑名称冲突的情况下可选择使用 `complex`。

3.4.8　其他类型

现在已经介绍完 C 语言的所有基本数据类型。有些人认为这些类型实在太多了，但有些人觉得还不够用。注意，虽然 C 语言没有字符串类型，但也能很好地处理字符串。第 4 章将详细介绍相关内容。

C 语言还有一些从基本类型衍生的其他类型，包括数组、指针、结构和联合。尽管后面章节中会详细介绍这些类型，但是本章的程序示例中已经用到了指针〔指针（*pointer*）指向变量或其他数据对象位置〕。例如，在 `scanf()` 函数中用到的前缀 `&`，便创建了一个指针，告诉 `scanf()` 把数据放在何处。

> **小结：基本数据类型**
>
> **关键字：**
>
> 基本数据类型由 11 个关键字组成：`int`、`long`、`short`、`unsigned`、`char`、`float`、`double`、`signed`、`_Bool`、`_Complex` 和`_Imaginary`。
>
> **有符号整型：**
>
> 有符号整型可用于表示正整数和负整数。
>
> - int ——系统给定的基本整数类型。C 语言规定 `int` 类型不小于 16 位。

- short 或 short int ——最大的 short 类型整数小于或等于最大的 int 类型整数。C 语言规定 short 类型至少占 16 位。
- long 或 long int ——该类型可表示的整数大于或等于最大的 int 类型整数。C 语言规定 long 类型至少占 32 位。
- long long 或 long long int ——该类型可表示的整数大于或等于最大的 long 类型整数。Long long 类型至少占 64 位。

一般而言，long 类型占用的内存比 short 类型大，int 类型的宽度要么和 long 类型相同，要么和 short 类型相同。例如，旧 DOS 系统的 PC 提供 16 位的 short 和 int，以及 32 位的 long；Windows 95 系统提供 16 位的 short 以及 32 位的 int 和 long。

无符号整型：

无符号整型只能用于表示零和正整数，因此无符号整型可表示的正整数比有符号整型的大。在整型类型前加上关键字 unsigned 表明该类型是无符号整型：unsignedint、unsigned long、unsigned short。单独的 unsigned 相当于 unsignedint。

字符类型：

可打印出来的符号（如 A、&和+）都是字符。根据定义，char 类型表示一个字符要占用 1 字节内存。出于历史原因，1 字节通常是 8 位，但是如果要表示基本字符集，也可以是 16 位或更大。

- char ——字符类型的关键字。有些编译器使用有符号的 char，而有些则使用无符号的 char。在需要时，可在 char 前面加上关键字 signed 或 unsigned 来指明具体使用哪一种类型。

布尔类型：

布尔值表示 true 和 false。C 语言用 1 表示 true，0 表示 false。

- _Bool ——布尔类型的关键字。布尔类型是无符号 int 类型，所占用的空间只要能储存 0 或 1 即可。

实浮点类型：

实浮点类型可表示正浮点数和负浮点数。

- float ——系统的基本浮点类型，可精确表示至少 6 位有效数字。
- double ——储存浮点数的范围（可能）更大，能表示比 float 类型更多的有效数字（至少 10 位，通常会更多）和更大的指数。
- long long ——储存浮点数的范围（可能）比 double 更大，能表示比 double 更多的有效数字和更大的指数。

复数和虚数浮点数：

虚数类型是可选的类型。复数的实部和虚部类型都基于实浮点类型来构成：

- float _Complex
- double _Complex
- long double _Complex
- float _Imaginary
- double _Imaginary
- long long _Imaginary

小结：如何声明简单变量

1. 选择需要的类型。
2. 使用有效的字符给变量起一个变量名。
3. 按以下格式进行声明：

```
类型说明符 变量名；
```

类型说明符由一个或多个关键字组成。下面是一些示例：

```
int erest;
unsigned short cash;
```

4. 可以同时声明相同类型的多个变量，用逗号分隔各变量名，如下所示：

```
char ch, init, ans;
```

5. 在声明的同时还可以初始化变量：

```
float mass = 6.0E24;
```

3.4.9　类型大小

如何知道当前系统的指定类型的大小是多少？运行程序清单 3.8，会列出当前系统的各类型的大小。

程序清单 3.8　typesize.c 程序

```
//* typesize.c -- 打印类型大小 */
#include <stdio.h>
int main(void)
{
    /* C99 为类型大小提供%zd 转换说明 */
    printf("Type int has a size of %zd bytes.\n", sizeof(int));
    printf("Type char has a size of %zd bytes.\n", sizeof(char));
    printf("Type long has a size of %zd bytes.\n", sizeof(long));
    printf("Type long long has a size of %zd bytes.\n",
            sizeof(long long));
    printf("Type double has a size of %zd bytes.\n",
           sizeof(double));
    printf("Type long double has a size of %zd bytes.\n",
           sizeof(long double));
    return 0;
}
```

sizeof 是 C 语言的内置运算符，以字节为单位给出指定类型的大小。C99 和 C11 提供%zd 转换说明匹配 sizeof 的返回类型[1]。一些不支持 C99 和 C11 的编译器可用%u 或%lu 代替%zd。

该程序的输出如下：

```
Type int has a size of 4 bytes.
Type char has a size of 1 bytes.
Type long has a size of 8 bytes.
Type long long has a size of 8 bytes.
Type double has a size of 8 bytes.
Type long double has a size of 16 bytes.
```

该程序列出了 6 种类型的大小，你也可以把程序中的类型更换成感兴趣的其他类型。注意，因为 C 语言定义了 char 类型是 1 字节，所以 char 类型的大小一定是 1 字节。而在 char 类型为 16 位、double

[1] 即，size_t 类型。——译者注

类型为 64 位的系统中，sizeof 给出的 double 是 4 字节。在 limits.h 和 float.h 头文件中有类型限制的相关信息（下一章将详细介绍这两个头文件）。

顺带一提，注意该程序最后几行 printf()语句都被分为两行，只要不在引号内部或一个单词中间断行，就可以这样写。

3.5 使用数据类型

编写程序时，应注意合理选择所需的变量及其类型。通常，用 int 或 float 类型表示数字，char 类型表示字符。在使用变量之前必须先声明，并选择有意义的变量名。初始化变量应使用与变量类型匹配的常数类型。例如：

```
int apples = 3;         /* 正确 */
int oranges = 3.0;      /* 不好的形式 */
```

与 Pascal 相比，C 在检查类型匹配方面不太严格。C 编译器甚至允许二次初始化，但在激活了较高级别警告时，会给出警告。最好不要养成这样的习惯。

把一个类型的数值初始化给不同类型的变量时，编译器会把值转换成与变量匹配的类型，这将导致部分数据丢失。例如，下面的初始化：

```
int cost = 12.99;       /* 用double类型的值初始化int类型的变量 */
float pi = 3.1415926536;    /* 用double类型的值初始化float类型的变量 */
```

第 1 个声明，cost 的值是 12。C 编译器把浮点数转换成整数时，会直接丢弃（截断）小数部分，而不进行四舍五入。第 2 个声明会损失一些精度，因为 C 只保证了 float 类型前 6 位的精度。编译器对这样的初始化可能给出警告。读者在编译程序清单 3.1 时可能就遇到了这种警告。

许多程序员和公司内部都有系统化的命名约定，在变量名中体现其类型。例如，用 i_前缀表示 int 类型，us_前缀表示 unsigned short 类型。这样，一眼就能看出来 i_smart 是 int 类型的变量，us_versmart 是 unsigned short 类型的变量。

3.6 参数和陷阱

有必要再次提醒读者注意 printf()函数的用法。读者应该还记得，传递给函数的信息被称为参数。例如，printf("Hello, pal.")函数调用有一个参数："Hello,pal."。双引号中的字符序列（如，"Hello,pal."）被称为字符串（*string*），第 4 章将详细讲解相关内容。现在，关键是要理解无论双引号中包含多少个字符和标点符号，一个字符串就是一个参数。

与此类似，scanf("%d", &weight)函数调用有两个参数："%d"和&weight。C 语言用逗号分隔函数中的参数。printf()和 scanf()函数与一般函数不同，它们的参数个数是可变的。例如，前面的程序示例中调用过带一个、两个，甚至三个参数的 printf()函数。程序要知道函数的参数个数才能正常工作。printf()和 scanf()函数用第 1 个参数表明后续有多少个参数，即第 1 个字符串中的转换说明与后面的参数一一对应。例如，下面的语句有两个%d 转换说明，说明后面还有两个参数：

```
printf("%d cats ate %d cans of tuna\n", cats, cans);
```

后面的确还有两个参数：cats 和 cans。

程序员要负责确保转换说明的数量、类型与后面参数的数量、类型相匹配。现在，C 语言通过函数原型机制检查函数调用时参数的个数和类型是否正确。但是，该机制对 printf()和 scanf()不起作用，因为这两个函数的参数个数可变。如果参数在匹配上有问题，会出现什么情况？假设你编写了程序清单 3.9

中的程序。

程序清单 3.9　badcount.c **程序**

```
/* badcount.c -- 参数错误的情况 */
#include <stdio.h>
int main(void)
{
    int n = 4;
    int m = 5;
    float f = 7.0f;
    float g = 8.0f;

    printf("%d\n", n, m);    /* 参数太多  */
    printf("%d %d %d\n", n); /* 参数太少  */
    printf("%d %d\n", f, g); /* 值的类型不匹配 */

    return 0;
}
```

XCode 4.6（OS 10.8）的输出如下：

```
4
4 1 -706337836
1606414344 1
```

Microsoft Visual Studio Express 2012（Windows 7）的输出如下：

```
4
4 0 0
0 1075576832
```

注意，用`%d`显示 `float` 类型的值，其值不会被转换成 `int` 类型。在不同的平台下，缺少参数或参数类型不匹配导致的结果不同。

所有编译器都能顺利编译并运行该程序，但其中大部分会给出警告。的确，有些编译器会捕获到这类问题，然而 C 标准对此未作要求。因此，计算机在运行时可能不会捕获这类错误。如果程序正常运行，很难觉察出来。如果程序没有打印出期望值或打印出意想不到的值，你才会检查 printf()函数中的参数个数和类型是否得当。

3.7　转义序列示例

再来看一个程序示例，该程序使用了一些特殊的转义序列。程序清单 3.10 演示了退格（`\b`）、水平制表符（`\t`）和回车（`\t`）的工作方式。这些概念在计算机使用电传打字机作为输出设备时就有了，但是它们不一定能与现代的图形接口兼容。例如，程序清单 3.10 在某些 Macintosh 的实现中就无法正常运行。

程序清单 3.10　escape.c **程序**

```
/* escape.c -- 使用转移序列 */
#include <stdio.h>
int main(void)
{
    float salary;

    printf("\aEnter your desired monthly salary:");  /* 1 */
    printf(" $_______\b\b\b\b\b\b\b");                /* 2 */
```

```
    scanf("%f", &salary);
    printf("\n\t$%.2f a month is $%.2f a year.", salary,
           salary * 12.0);                              /* 3 */
    printf("\rGee!\n");                                 /* 4 */

    return 0;
}
```

3.7.1 程序运行情况

假设在系统中运行的转义序列行为与本章描述的行为一致（实际行为可能不同。例如，XCode 4.6 把\a、\b 和\r 显示为颠倒的问号），下面我们来分析这个程序。

第 1 条 printf()语句（注释中标为 1）发出一声警报（因为使用了\a），然后打印下面的内容：

```
Enter your desired monthly salary:
```

因为 printf()中的字符串末尾没有\n，所以光标停留在冒号后面。

第 2 条 printf()语句在光标处接着打印，屏幕上显示的内容是：

```
Enter your desired monthly salary: $_______
```

冒号和美元符号之间有一个空格，这是因为第 2 条 printf()语句中的字符串以一个空格开始。7 个退格字符使得光标左移 7 个位置，即把光标移至 7 个下划线字符的前面，紧跟在美元符号后面。通常，退格不会擦除退回所经过的字符，但有些实现是擦除的，这和本例不同。

假设键入的数据是 4000.00（并按下 Enter 键），屏幕显示的内容应该是：

```
Enter your desired monthly salary: $4000.00
```

键入的字符替换了下划线字符。按下 Enter 键后，光标移至下一行的起始处。

第 3 条 printf()语句中的字符串以\n\t 开始。换行字符使光标移至下一行起始处。水平制表符使光标移至该行的下一个制表点，一般是第 9 列（但不一定）。然后打印字符串中的其他内容。执行完该语句后，此时屏幕显示的内容应该是：

```
Enter your desired monthly salary: $4000.00
        $4000.00 a month is $48000.00 a year.
```

因为这条 printf()语句中没有使用换行字符，所以光标停留在最后的点号后面。

第 4 条 printf()语句以\r 开始。这使得光标回到当前行的起始处。然后打印 Gee!，接着\n 使光标移至下一行的起始处。屏幕最后显示的内容应该是：

```
Enter your desired monthly salary: $4000.00
Gee! $4000.00 a month is $48000.00 a year.
```

3.7.2 刷新输出

printf()何时把输出发送到屏幕上？最初，printf()语句把输出发送到一个叫作缓冲区（*buffer*）的中间存储区域，然后缓冲区中的内容再不断被发送到屏幕上。C 标准明确规定了何时把缓冲区中的内容发送到屏幕：当缓冲区满、遇到换行字符或需要输入的时候（从缓冲区把数据发送到屏幕或文件被称为*刷新缓冲区*）。例如，前两个 printf()语句既没有填满缓冲区，也没有换行符，但是下一条 scanf()语句要求用户输入，这迫使 printf()的输出被发送到屏幕上。

旧式编译器遇到 scanf()也不会强行刷新缓冲区，程序会停在那里不显示任何提示内容，等待用户输入数据。在这种情况下，可以使用换行字符刷新缓冲区。代码应改为：

```
printf("Enter your desired monthly salary:\n");
```

```
scanf("%f", &salary);
```

无论接下来的输入是否能刷新缓冲区，代码都会正常运行。这将导致光标移至下一行起始处，用户无法在提示内容同一行输入数据。还有一种刷新缓冲区的方法是使用 fflush()函数，详见第 13 章。

3.8 关键概念

C 语言提供了大量的数值类型，目的是为程序员提供方便。那以整数类型为例，C 认为一种整型不够，提供了有符号、无符号，以及大小不同的整型，以满足不同程序的需求。

计算机中的浮点数和整数在本质上不同，其存储方式和运算过程有很大区别。即使两个 32 位存储单元储存的位组合完全相同，但是一个解释为 float 类型，另一个解释为 long 类型，这两个相同的位组合表示的值也完全不同。例如，在 PC 中，假设一个位组合表示 float 类型的数 256.0，如果将其解释为 long 类型，得到的值是 113246208。C 语言允许编写混合数据类型的表达式，但是会进行自动类型转换，以便在实际运算时统一使用一种类型。

计算机在内存中用数值编码来表示字符。美国最常用的是 ASCII 码，除此之外 C 也支持其他编码。字符常量是计算机系统使用的数值编码的符号表示，它表示为单引号括起来的字符，如'A'。

3.9 本章小结

C 有多种的数据类型。基本数据类型分为两大类：整数类型和浮点数类型。通过为类型分配的储存量以及是有符号还是无符号，区分不同的整数类型。最小的整数类型是 char，因实现不同，可以是有符号的 char 或无符号的 char，即 unsigned char 或 signed char。但是，通常用 char 类型表示小整数时才这样显示说明。其他整数类型有 short、int、long 和 long long 类型。C 规定，后面的类型不能小于前面的类型。上述都是有符号类型，但也可以使用 unsigned 关键字创建相应的无符号类型：unsigned short、unsigned int、unsigned long 和 unsigned long long。或者，在类型名前加上 signed 修饰符显式表明该类型是有符号类型。最后，_Bool 类型是一种无符号类型，可储存 0 或 1，分别代表 false 和 true。

浮点类型有 3 种：float、double 和 C90 新增的 long double。后面的类型应大于或等于前面的类型。有些实现可选择支持复数类型和虚数类型，通过关键字_Complex 和_Imaginary 与浮点类型的关键字组合（如，double _Complex 类型和 float _Imaginary 类型）来表示这些类型。

整数可以表示为十进制、八进制或十六进制。0 前缀表示八进制数，0x 或 0X 前缀表示十六进制数。例如，32、040、0x20 分别以十进制、八进制、十六进制表示同一个值。l 或 L 前缀表明该值是 long 类型，ll 或 LL 前缀表明该值是 long long 类型。

在 C 语言中，直接表示一个字符常量的方法是：把该字符用单引号括起来，如'Q'、'8'和'$'。C 语言的转义序列（如，'\n'）表示某些非打印字符。另外，还可以在八进制或十六进制数前加上一个反斜杠（如，'\007'），表示 ASCII 码中的一个字符。

浮点数可写成固定小数点的形式（如，9393.912）或指数形式（如，7.38E10）。C99 和 C11 提供了第 3 种指数表示法，即用十六进制数和 2 的幂来表示（如，0xa.1fp10）。

printf()函数根据转换说明打印各种类型的值。转换说明最简单的形式由一个百分号（%）和一个转换字符组成，如%d 或%f。

3.10 复习题

复习题的参考答案在附录 A 中。

1．指出下面各种数据使用的合适数据类型（有些可使用多种数据类型）：

a．East Simpleton 的人口

b．DVD 影碟的价格

c．本章出现次数最多的字母

d．本章出现次数最多的字母次数

2．在什么情况下要用 long 类型的变量代替 int 类型的变量？

3．使用哪些可移植的数据类型可以获得 32 位有符号整数？选择的理由是什么？

4．指出下列常量的类型和含义（如果有的话）：

a．`'\b'`

b．`1066`

c．`99.44`

d．`0XAA`

e．`2.0e30`

5．Dottie Cawm 编写了一个程序，请找出程序中的错误。

```
include <stdio.h>
main
(
    float g; h;
    float tax, rate;

    g = e21;
    tax = rate*g;
)
```

6．写出下列常量在声明中使用的数据类型和在 `printf()` 中对应的转换说明：

常量	类型	转换说明（%转换字符）
12		
0X3		
'C'		
2.34E07		
'\040'		
7.0		
6L		
6.0f		
0x5.b6p12		

7．写出下列常量在声明中使用的数据类型和在 `printf()` 中对应的转换说明（假设 `int` 为 16 位）：

常量	类型	转换说明（%转换字符）
012		
2.9e05L		
's'		
100000		
'\n'		
20.0f		
0x44		
-40		

8．假设程序的开头有下列声明：

```
int imate = 2;
long shot = 53456;
char grade = 'A';
float log = 2.71828;
```

把下面 printf()语句中的转换字符补充完整：

```
printf("The odds against the %__ were %__ to 1.\n", imate, shot);
printf("A score of %__ is not an %__ grade.\n", log, grade);
```

9．假设 ch 是 char 类型的变量。分别使用转义序列、十进制值、八进制字符常量和十六进制字符常量把回车字符赋给 ch（假设使用 ASCII 编码值）。

10．修正下面的程序（在 C 中，/表示除以）。

```
void main(int) / this program is perfect /
{
    cows, legs integer;
    printf("How many cow legs did you count?\n);
    scanf("%c", legs);
    cows = legs / 4;
    printf("That implies there are %f cows.\n", cows)
}
```

11．指出下列转义序列的含义：

a．\n

b．\\

c．\"

d．\t

3.11　编程练习

1．通过试验（即编写带有此类问题的程序）观察系统如何处理整数上溢、浮点数上溢和浮点数下溢的情况。

2．编写一个程序，要求提示输入一个 ASCII 码值（如，66），然后打印输入的字符。

3．编写一个程序，发出一声警报，然后打印下面的文本：

```
Startled by the sudden sound, Sally shouted,
"By the Great Pumpkin, what was that!"
```

4．编写一个程序，读取一个浮点数，先打印成小数点形式，再打印成指数形式。然后，如果系统支持，

再打印成 p 记数法（即十六进制记数法）。按以下格式输出（实际显示的指数位数因系统而异）：

```
Enter a floating-point value: 64.25
fixed-point notation: 64.250000
exponential notation: 6.425000e+01
p notation: 0x1.01p+6
```

5．一年大约有 3.156×10^{7} 秒。编写一个程序，提示用户输入年龄，然后显示该年龄对应的秒数。

6．1 个水分子的质量约为 3.0×10^{-23} 克。1 夸脱水大约是 950 克。编写一个程序，提示用户输入水的夸脱数，并显示水分子的数量。

7．1 英寸相当于 2.54 厘米。编写一个程序，提示用户输入身高（/英寸），然后以厘米为单位显示身高。

8．在美国的体积测量系统中，1 品脱等于 2 杯，1 杯等于 8 盎司，1 盎司等于 2 大汤勺，1 大汤勺等于 3 茶勺。编写一个程序，提示用户输入杯数，并以品脱、盎司、汤勺、茶勺为单位显示等价容量。思考对于该程序，为何使用浮点类型比整数类型更合适？

第 4 章

字符串和格式化输入/输出

本章介绍以下内容：

- 函数：strlen()
- 关键字：const
- 字符串
- 如何创建、存储字符串
- 如何使用 strlen()函数获取字符串的长度
- 用 C 预处理器指令#define 和 ANSIC 的 const 修饰符创建符号常量

本章重点介绍输入和输出。与程序交互和使用字符串可以编写个性化的程序，本章将详细介绍 C 语言的两个输入/输出函数：printf()和 scanf()。学会使用这两个函数，不仅能与用户交互，还可根据个人喜好和任务要求格式化输出。最后，简要介绍一个重要的工具——C 预处理器指令，并学习如何定义、使用符号常量。

4.1 前导程序

与前两章一样，本章以一个简单的程序开始。程序清单 4.1 与用户进行简单的交互。为了使程序的形式灵活多样，代码中使用了新的注释风格。

程序清单 4.1　talkback.c 程序

```
// talkback.c -- 演示与用户交互
#include <stdio.h>
#include <string.h>     // 提供strlen()函数的原型
#define DENSITY 62.4    // 人体密度（单位：磅/立方英尺）
int main()
{
    float weight, volume;
    int size, letters;
    char name[40];       // name是一个可容纳40个字符的数组

    printf("Hi! What's your first name?\n");
    scanf("%s", name);
    printf("%s, what's your weight in pounds?\n", name);
    scanf("%f", &weight);
    size = sizeof name;
    letters = strlen(name);
    volume = weight / DENSITY;
    printf("Well, %s, your volume is %2.2f cubic feet.\n",
            name, volume);
    printf("Also, your first name has %d letters,\n",
            letters);
```

```
    printf("and we have %d bytes to store it.\n", size);

    return 0;
}
```

运行 talkback.c 程序，输入结果如下：

```
Hi! What's your first name?
Christine
Christine, what's your weight in pounds?
154
Well, Christine, your volume is 2.47 cubic feet.
Also, your first name has 9 letters,
and we have 40 bytes to store it.
```

该程序包含以下新特性。

- 用**数组**（*array*）储存**字符串**（*character string*）。在该程序中，用户输入的名被储存在数组中，该数组占用内存中 40 个连续的字节，每个字节储存一个字符值。
- 使用**%s 转换说明**来处理字符串的输入和输出。注意，在 scanf()中，name 没有&前缀，而 weight 有（稍后解释，&weight 和 name 都是地址）。
- 用 C 预处理器把字符常量 DENSITY 定义为 62.4。
- 用 C 函数 strlen()获取字符串的长度。

对于 BASIC 的输入/输出而言，C 的输入/输出看上去有些复杂。不过，复杂换来的是程序的高效和方便控制输入/输出。而且，一旦熟悉用法后，会发现它很简单。

4.2　字符串简介

字符串（*character string*）是一个或多个字符的序列，如下所示：

```
"Zing went the strings of my heart!"
```

双引号不是字符串的一部分。双引号仅告知编译器它括起来的是字符串，正如单引号用于标识单个字符一样。

4.2.1　char 类型数组和 null 字符

C 语言没有专门用于储存字符串的变量类型，字符串都被储存在 char 类型的数组中。数组由连续的存储单元组成，字符串中的字符被储存在相邻的存储单元中，每个单元储存一个字符（见图 4.1）。

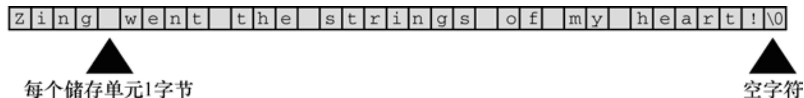


图 4.1　数组中的字符串

注意图 4.1 中数组末尾位置的字符\0。这是**空字符**（*null character*），C 语言用它标记字符串的结束。空字符不是数字 0，它是非打印字符，其 ASCII 码值是（或等价于）0。C 中的字符串一定以空字符结束，这意味着数组的容量必须至少比待存储字符串中的字符数多 1。因此，程序清单 4.1 中有 40 个存储单元的字符串，只能储存 39 个字符，剩下一个字节留给空字符。

那么，什么是数组？可以把数组看作是一行连续的多个存储单元。用更正式的说法是，数组是同类型

数据元素的有序序列。程序清单 4.1 通过以下声明创建了一个包含 40 个存储单元（或元素）的数组，每个单元储存一个 char 类型的值：

```
char name[40];
```

name 后面的方括号表明这是一个数组，方括号中的 40 表明该数组中的元素数量。char 表明每个元素的类型（见图 4.2）。

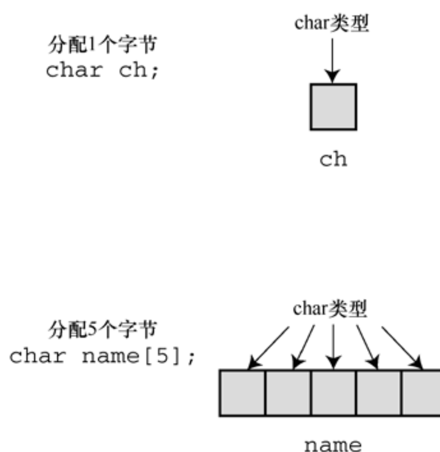


图 4.2　声明一个变量和声明一个数组

字符串看上去比较复杂！必须先创建一个数组，把字符串中的字符逐个放入数组，还要记得在末尾加上一个\0。还好，计算机可以自己处理这些细节。

4.2.2　使用字符串

试着运行程序清单 4.2，使用字符串其实很简单。

程序清单 4.2　praise1.c 程序

```
/* praise1.c -- 使用不同类型的字符串 */
#include <stdio.h>
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];

    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);

    return 0;
}
```

%s 告诉 printf()打印一个字符串。%s 出现了两次，因为程序要打印两个字符串：一个储存在 name 数组中；一个由 PRAISE 来表示。运行 praise1.c，其输出如下所示：

```
What's your name? Angela Plains
Hello, Angela. You are an extraordinary being.
```

你不用亲自把空字符放入字符串末尾，scanf()在读取输入时就已完成这项工作。也不用在**字符串常**

量 PRAISE 末尾添加空字符。稍后我们会解释#define 指令，现在先理解 PRAISE 后面用双引号括起来的文本是一个字符串。编译器会在末尾加上空字符。

注意（这很重要），scanf()只读取了 Angela Plains 中的 Angela，它在遇到第 1 个空白（空格、制表符或换行符）时就不再读取输入。因此，scanf()在读到 Angela 和 Plains 之间的空格时就停止了。一般而言，根据%s 转换说明，scanf()只会读取字符串中的一个单词，而不是一整句。C 语言还有其他的输入函数（如，fgets()），用于读取一般字符串。后面章节将详细介绍这些函数。

字符串和字符

字符串常量"x"和字符常量'x'不同。区别之一在于'x'是基本类型（char），而"x"是派生类型（char 数组）；区别之二是"x"实际上由两个字符组成：'x'和空字符\0（见图 4.3）。

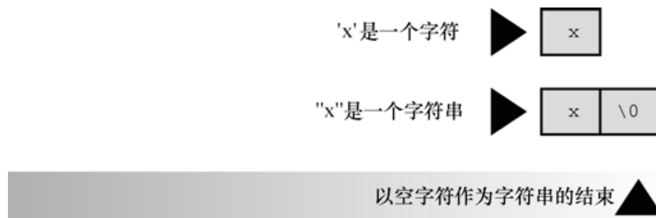


图 4.3　字符'x'和字符串"x"

4.2.3　strlen()函数

上一章提到了 sizeof 运算符，它以字节为单位给出对象的大小。strlen()函数给出字符串中的字符长度。因为 1 字节储存一个字符，读者可能认为把两种方法应用于字符串得到的结果相同，但事实并非如此。请根据程序清单 4.3，在程序清单 4.2 中添加几行代码，看看为什么会这样。

程序清单 4.3　praise2.c 程序

```
/* praise2.c */
// 如果编译器不识别%zd，尝试换成%u 或%lu。
#include <stdio.h>
#include <string.h>     /* 提供 strlen()函数的原型 */
#define PRAISE "You are an extraordinary being."
int main(void)
{
    char name[40];

    printf("What's your name? ");
    scanf("%s", name);
    printf("Hello, %s. %s\n", name, PRAISE);
    printf("Your name of %zd letters occupies %zd memory cells.\n",
            strlen(name), sizeof name);
    printf("The phrase of praise has %zd letters ",
            strlen(PRAISE));
    printf("and occupies %zd memory cells.\n", sizeof PRAISE);

    return 0;
}
```

如果使用 ANSI C 之前的编译器，必须移除这一行：

```
#include <string.h>
```

string.h 头文件包含多个与字符串相关的函数原型，包括 strlen()。第 11 章将详细介绍该头文件（顺带一提，一些 ANSI 之前的 UNIX 系统用 strings.h 代替 string.h，其中也包含了一些字符串函数的声明）。

一般而言，C 把函数库中相关的函数归为一类，并为每类函数提供一个头文件。例如，printf()和 scanf()都隶属标准输入和输出函数，使用 stdio.h 头文件。string.h 头文件中包含了 strlen()函数和其他一些与字符串相关的函数（如拷贝字符串的函数和字符串查找函数）。

注意，程序清单 4.3 使用了两种方法处理很长的 printf()语句。第 1 种方法是将 printf()语句分为两行（可以在参数之间断为两行，但是不要在双引号中的字符串中间断开）；第 2 种方法是使用两个 printf()语句打印一行内容，只在第 2 条 printf()语句中使用换行符（\n）。运行该程序，其交互输出如下：

```
What's your name? Serendipity Chance
Hello, Serendipity. You are an extraordinary being.
Your name of 11 letters occupies 40 memory cells.
The phrase of praise has 31 letters and occupies 32 memory cells.
```

sizeof 运算符报告，name 数组有 40 个存储单元。但是，只有前 11 个单元用来储存 Serendipity，所以 strlen()得出的结果是 11。name 数组的第 12 个单元储存空字符，strlen()并未将其计入。图 4.4 演示了这个概念。

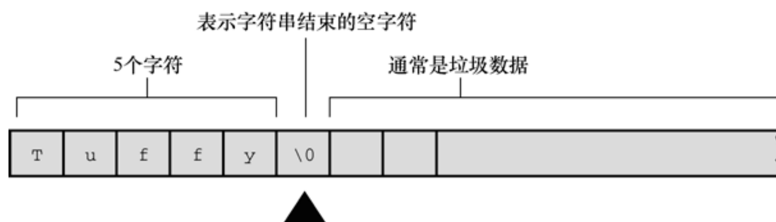


图 4.4 strlen()函数知道在何处停止

对于 PRAISE，用 strlen()得出的也是字符串中的字符数（包括空格和标点符号）。然而，sizeof 运算符给出的数更大，因为它把字符串末尾不可见的空字符也计算在内。该程序并未明确告诉计算机要给字符串预留多少空间，所以它必须计算双引号内的字符数。

第 3 章提到过，C99 和 C11 标准专门为 sizeof 运算符的返回类型添加了%zd 转换说明，这对于 strlen()同样适用。对于早期的 C，还要知道 sizeof 和 strlen()返回的实际类型（通常是 unsigned 或 unsigned long）。

另外，还要注意一点：上一章的 sizeof 使用了圆括号，但本例没有。圆括号的使用时机否取决于运算对象是类型还是特定量？运算对象是类型时，圆括号必不可少，但是对于特定量，可有可无。也就是说，对于类型，应写成 sizeof(char)或 sizeof(float)；对于特定量，可写成 sizeof name 或 sizeof 6.28。尽管如此，还是建议所有情况下都使用圆括号，如 sizeof(6.28)。

程序清单 4.3 中使用 strlen()和 sizeof，完全是为了满足读者的好奇心。在实际应用中，strlen()和 sizeof 是非常重要的编程工具。例如，在各种要处理字符串的程序中，strlen()很有用。详见第 11 章。

下面我们来学习#define 指令。

4.3　常量和 C 预处理器

有时，在程序中要使用常量。例如，可以这样计算圆的周长：

```
circumference = 3.14159 * diameter;
```

这里，常量 3.14159 代表著名的常量 pi（π）。在该例中，输入实际值便可使用这个常量。然而，这种情况使用**符号常量**（*symbolic constant*）会更好。也就是说，使用下面的语句，计算机稍后会用实际值完成替换：

```
circumference = pi * diameter;
```

为什么使用符号常量更好？首先，常量名比数字表达的信息更多。请比较以下两条语句：

```
owed = 0.015 * housevalue;
owed = taxrate * housevalue;
```

如果阅读一个很长的程序，第 2 条语句所表达的含义更清楚。

另外，假设程序中的多处使用一个常量，有时需要改变它的值。毕竟，税率通常是浮动的。如果程序使用符号常量，则只需更改符号常量的定义，不用在程序中查找使用常量的地方，然后逐一修改。

那么，如何创建符号常量？方法之一是声明一个变量，然后将该变量设置为所需的常量。可以这样写：

```
float taxrate;
taxrate = 0.015;
```

这样做提供了一个符号名，但是 taxrate 是一个变量，程序可能会无意间改变它的值。C 语言还提供了一个更好的方案——C 预处理器。第 2 章中介绍了预处理器如何使用#include 包含其他文件的信息。预处理器也可用来定义常量。只需在程序顶部添加下面一行：

```
#define TAXRATE 0.015
```

编译程序时，程序中所有的TAXRATE都会被替换成 0.015。这一过程被称为**编译时替换**（*compile-time substitution*）。在运行程序时，程序中所有的替换均已完成（见图 4.5）。通常，这样定义的常量也称为**明示常量**（*manifest constant*）[1]。

请注意格式，首先是#define，接着是符号常量名（TAXRATE），然后是符号常量的值（0.015）（注意，其中并没有=符号）。所以，其通用格式如下：

```
#define NAME value
```

实际应用时，用选定的符号常量名和合适的值来替换 *NAME* 和 *value*。注意，末尾不用加分号，因为这是一种由预处理器处理的替换机制。为什么 TAXRATE 要用大写？用大写表示符号常量是 C 语言一贯的传统。这样，在程序中看到全大写的名称就立刻明白这是一个符号常量，而非变量。大写常量只是为了提高程序的可读性，即使全用小写来表示符号常量，程序也能照常运行。尽管如此，初学者还是应该养成大写常量的好习惯。

另外，还有一个不常用的命名约定，即在名称前带 c_或 k_前缀来表示常量（如，c_level 或 k_line）。

符号常量的命名规则与变量相同。可以使用大小写字母、数字和下划线字符，首字符不能为数字。程序清单 4.4 演示了一个简单的示例。

[1] 其实，符号常量的概念在 K&R 合著的《C 语言程序设计》中介绍过。但是，在历年的 C 标准中（包括最新的 C11），并没有符号常量的概念，只提到过#define 最简单的用法是定义一个“明示常量”。市面上各编程书籍对此概念的理解不同，有些作者把#define 宏定义实现的“常量”归为“明示常量”；有些作者（如，本书的作者）则认为“明示常量”相当于“符号常量”。——译者注

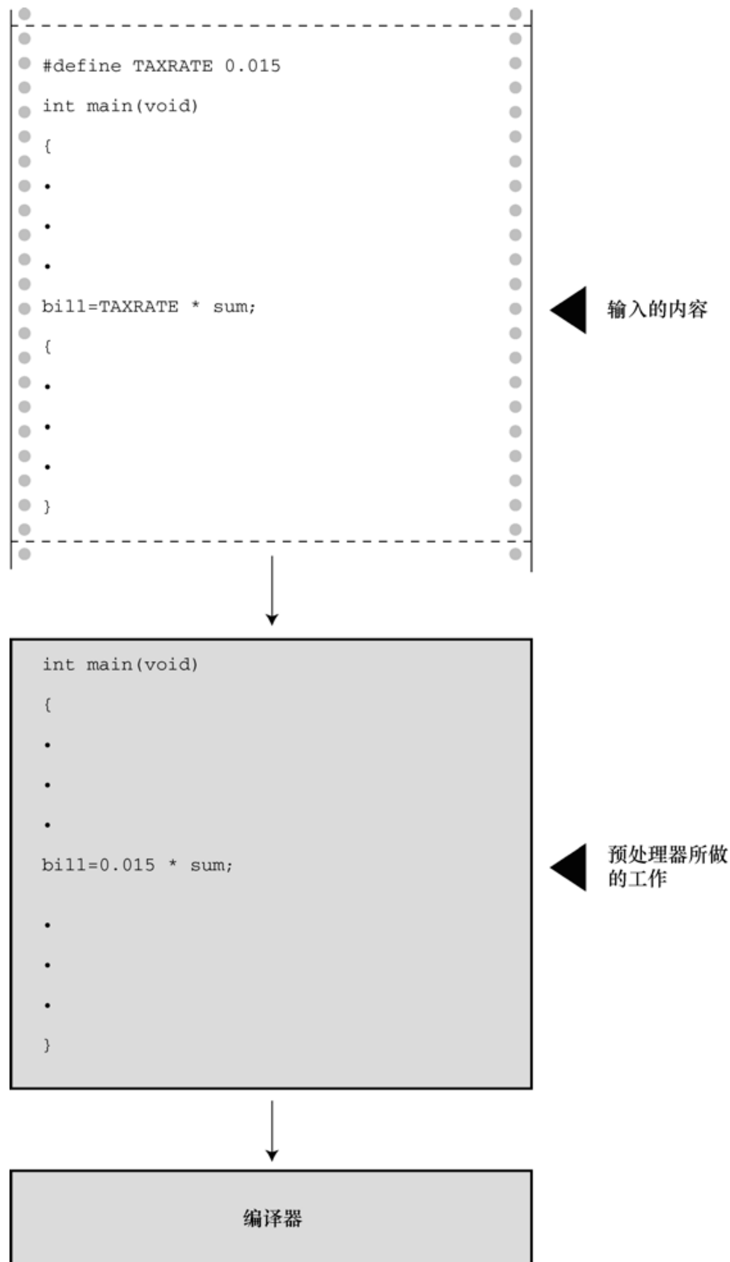


图 4.5　输入的内容和编译后的内容

程序清单 4.4　pizza.c 程序

```
/* pizza.c -- 在比萨饼程序中使用已定义的常量 */
#include <stdio.h>
#define PI 3.14159
int main(void)
{
    float area, circum, radius;

    printf("What is the radius of your pizza?\n");
    scanf("%f", &radius);
```

```
    area = PI * radius * radius;
    circum = 2.0 * PI *radius;
    printf("Your basic pizza parameters are as follows:\n");
    printf("circumference = %1.2f, area = %1.2f\n", circum,area);

    return 0;
}
```

printf()语句中的%1.2f 表明，结果被四舍五入为两位小数输出。下面是一个输出示例：

```
What is the radius of your pizza?
6.0
Your basic pizza parameters are as follows:
circumference = 37.70, area = 113.10
```

#define 指令还可定义字符和字符串常量。前者使用单引号，后者使用双引号。如下所示：

```
#define BEEP '\a'
#define TEE 'T'
#define ESC '\033'
#define OOPS "Now you have done it!"
```

记住，符号常量名后面的内容被用来替换符号常量。不要犯这样的常见错误：

```
/* 错误的格式 */
#define TOES = 20
```

如果这样做，替换 TOES 的是= 20，而不是 20。这种情况下，下面的语句：

```
digits = fingers + TOES;
```

将被转换成错误的语句：

```
digits = fingers + = 20;
```

4.3.1　const 限定符

C90 标准新增了const关键字，用于限定一个变量为只读[1]。其声明如下：

```
const int MONTHS = 12; // MONTHS 在程序中不可更改，值为 12
```

这使得 MONTHS 成为一个只读值。也就是说，可以在计算中使用 MONTHS，可以打印 MONTHS，但是不能更改 MONTHS 的值。const 用起来比#define 更灵活，第 12 章将讨论与 const 相关的内容。

4.3.2　明示常量

C头文件limits.h和float.h分别提供了与整数类型和浮点类型大小限制相关的详细信息。每个头文件都定义了一系列供实现使用的明示常量[2]。例如，limits.h头文件包含以下类似的代码：

```
#define INT_MAX +32767
#define INT_MIN -32768
```

这些明示常量代表 int 类型可表示的最大值和最小值。如果系统使用 32 位的 int，该头文件会为这些明示常量提供不同的值。如果在程序中包含 limits.h 头文件，就可编写下面的代码：

```
printf("Maximum int value on this system = %d\n", INT_MAX);
```

如果系统使用 4 字节的 int，limits.h 头文件会提供符合 4 字节 int 的 INT_MAX 和 INT_MIN。表 4.1 列出了 limits.h 中能找到的一些明示常量。

---

[1] 注意，在 C 语言中，用 const 类型限定符声明的是变量，不是常量。——译者注
[2] 再次提醒读者注意，本书作者认为 “ 明示常量 ” 相当于 “ 符号常量 ”，经常在书中混用这两个术语。——译者注

表 4.1 limits.h 中的一些明示常量

明示常量	含义
CHAR_BIT	char 类型的位数
CHAR_MAX	char 类型的最大值
CHAR_MIN	char 类型的最小值
SCHAR_MAX	signed char 类型的最大值
SCHAR_MIN	signed char 类型的最小值
UCHAR_MAX	unsigned char 类型的最大值
SHRT_MAX	short 类型的最大值
SHRT_MIN	short 类型的最小值
USHRT_MAX	unsigned short 类型的最大值
INT_MAX	int 类型的最大值
INT_MIN	int 类型的最小值
UINT_MAX	unsigned int 的最大值
LONG_MAX	long 类型的最大值
LONG_MIN	long 类型的最小值
ULONG_MAX	unsigned long 类型的最大值
LLONG_MAX	long long 类型的最大值
LLONG_MIN	long long 类型的最小值
ULLONG_MAX	unsigned long long 类型的最大值

类似地，float.h 头文件中也定义一些明示常量，如 FLT_DIG 和 DBL_DIG，分别表示 float 类型和 double 类型的有效数字位数。表 4.2 列出了 float.h 中的一些明示常量（可以使用文本编辑器打开并查看系统使用的 float.h 头文件）。表中所列都与 float 类型相关。把明示常量名中的 FLT 分别替换成 DBL 和 LDBL，即可分别表示 double 和 long double 类型对应的明示常量（表中假设系统使用 2 的幂来表示浮点数）。

表 4.2 float.h 中的一些明示常量

明示常量	含义
FLT_MANT_DIG	float 类型的尾数位数
FLT_DIG	float 类型的最少有效数字位数（十进制）
FLT_MIN_10_EXP	带全部有效数字的 float 类型的最小负指数（以 10 为底）
FLT_MAX_10_EXP	float 类型的最大正指数（以 10 为底）
FLT_MIN	保留全部精度的 float 类型最小正数
FLT_MAX	float 类型的最大正数
FLT_EPSILON	1.00 和比 1.00 大的最小 float 类型值之间的差值

程序清单 4.5 演示了如何使用 float.h 和 limits.h 中的数据（注意，编译器要完全支持 C99 标准才能识别 LLONG_MIN 标识符）。

程序清单 4.5　defines.c 程序

```
// defines.c -- 使用 limit.h 和 float 头文件中定义的明示常量
#include <stdio.h>
#include <limits.h>    // 整型限制
#include <float.h>     // 浮点型限制
int main(void)
{
    printf("Some number limits for this system:\n");
    printf("Biggest int: %d\n", INT_MAX);
    printf("Smallest long long: %lld\n", LLONG_MIN);
    printf("One byte = %d bits on this system.\n", CHAR_BIT);
    printf("Largest double: %e\n", DBL_MAX);
    printf("Smallest normal float: %e\n", FLT_MIN);
    printf("float precision = %d digits\n", FLT_DIG);
    printf("float epsilon = %e\n", FLT_EPSILON);

    return 0;
}
```

该程序的输出示例如下：

```
Some number limits for this system:
Biggest int: 2147483647
Smallest long long: -9223372036854775808
One byte = 8 bits on this system.
Largest double: 1.797693e+308
Smallest normal float: 1.175494e-38
float precision = 6 digits
float epsilon = 1.192093e-07
```

C 预处理器是非常有用的工具，要好好利用它。本书的后面章节中会介绍更多相关应用。

4.4　printf()和 scanf()

printf()函数和 scanf()函数能让用户可以与程序交流，它们是**输入/输出函数**，或简称为 *I/O* **函数**。它们不仅是 C 语言中的 I/O 函数，而且是最多才多艺的函数。过去，这些函数和 C 库的一些其他函数一样，并不是 C 语言定义的一部分。最初，C 把输入/输出的实现留给了编译器的作者，这样可以针对特殊的机器更好地匹配输入/输出。后来，考虑到兼容性的问题，各编译器都提供不同版本的 printf()和 scanf()。尽管如此，各版本之间偶尔有一些差异。C90 和 C99 标准规定了这些函数的标准版本，本书亦遵循这一标准。

虽然 printf()是输出函数，scanf()是输入函数，但是它们的工作原理几乎相同。两个函数都使用格式字符串和参数列表。我们先介绍 printf()，再介绍 scanf()。

4.4.1　printf()函数

请求 printf()函数打印数据的指令要与待打印数据的类型相匹配。例如，打印整数时使用%d，打印字符时使用%c。这些符号被称为**转换说明**（*conversion specification*），它们指定了如何把数据转换成可显示的形式。我们先列出 ANSI C 标准为 printf()提供的转换说明，然后再示范如何使用一些较常见的转换说明。表 4.3 列出了一些转换说明和各自对应的输出类型。

表 4.3 转换说明及其打印的输出结果

转换说明	输出
%a	浮点数、十六进制数和 p 记数法（C99/C11）
%A	浮点数、十六进制数和 p 记数法（C99/C11）
%c	单个字符
%d	有符号十进制整数
%e	浮点数，e 记数法
%E	浮点数，e 记数法
%f	浮点数，十进制记数法
%g	根据值的不同，自动选择%f 或%e。%e 格式用于指数小于-4 或者大于或等于精度时
%G	根据值的不同，自动选择%f 或%E。%E 格式用于指数小于-4 或者大于或等于精度时
%i	有符号十进制整数（与%d 相同）
%o	无符号八进制整数
%p	指针
%s	字符串
%u	无符号十进制整数
%x	无符号十六进制整数，使用十六进制数 0f
%X	无符号十六进制整数，使用十六进制数 0F
%%	打印一个百分号

4.4.2 使用 printf()

程序清单 4.6 的程序中使用了一些转换说明。

程序清单 4.6 printout.c 程序

```
/* printout.c -- 使用转换说明 */
#include <stdio.h>
#define PI 3.141593
int main(void)
{
    int number = 7;
    float pies = 12.75;
    int cost = 7800;


    printf("The %d contestants ate %f berry pies.\n", number,
            pies);
    printf("The value of pi is %f.\n", PI);
    printf("Farewell! thou art too dear for my possessing,\n");
    printf("%c%d\n", '$', 2 * cost);

    return 0;
}
```

该程序的输出如下：

```
The 7 contestants ate 12.750000 berry pies.
The value of pi is 3.141593.
Farewell! thou art too dear for my possessing,
$15600
```

这是 printf()函数的格式：

```
printf( 格式字符串, 待打印项 1, 待打印项 2,...);
```

待打印项 1、待打印项 2 等都是要打印的项。它们可以是变量、常量，甚至是在打印之前先要计算的表达式。第 3 章提到过，格式字符串应包含每个待打印项对应的转换说明。例如，考虑下面的语句：

```
printf("The %d contestants ate %f berry pies.\n", number,pies);
```

格式字符串是双引号括起来的内容。上面语句的格式字符串包含了两个待打印项 number 和 poes 对应的两个转换说明。图 4.6 演示了 printf()语句的另一个例子。

下面是程序清单 4.6 中的另一行：

```
printf("The value of pi is %f.\n", PI);
```

该语句中，待打印项列表只有一个项——符号常量 PI。

如图 4.7 所示，格式字符串包含两种形式不同的信息：

- 实际要打印的字符；
- 转换说明。



图 4.6　printf()的参数

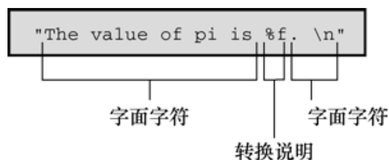


图 4.7　剖析格式字符串

> **警告**
>
> 格式字符串中的转换说明一定要与后面的每个项相匹配，若忘记这个基本要求会导致严重的后果。千万别写成下面这样：
>
> ```
> printf("The score was Squids %d, Slugs %d.\n", score1);
> ```
>
> 这里，第 2 个%d 没有对应任何项。系统不同，导致的结果也不同。不过，出现这种问题最好的状况是得到无意义的值。

如果只打印短语或句子，就不需要使用任何转换说明。如果只打印数据，也不用加入说明文字。程序清单 4.6 中的最后两个 printf()语句都没问题：

```
printf("Farewell! thou art too dear for my possessing,\n");
printf("%c%d\n", '$', 2 * cost);
```

注意第 2 条语句，待打印列表的第 1 个项是一个字符常量，不是变量；第 2 个项是一个乘法表达式。这说明 printf()使用的是值，无论是变量、常量还是表达式的值。

由于 printf()函数使用%符号来标识转换说明，因此打印%符号就成了个问题。如果单独使用一个%符号，编译器会认为漏掉了一个转换字符。解决方法很简单，使用两个%符号就行了：

```
pc = 2*6;
printf("Only %d%% of Sally's gribbles were edible.\n", pc);
```

下面是输出结果：

```
Only 12% of Sally's gribbles were edible.
```

4.4.3　printf()的转换说明修饰符

在%和转换字符之间插入修饰符可修饰基本的转换说明。表 4.4 和表 4.5 列出可作为修饰符的合法字符。如果要插入多个字符，其书写顺序应该与表 4.4 中列出的顺序相同。不是所有的组合都可行。表中有些字符是 C99 新增的，如果编译器不支持 C99，则可能不支持表中的所有项。

表 4.4　printf()的修饰符

修饰符	含义
标记	表 4.5 描述了 5 种标记（-、+、空格、#和 0），可以不使用标记或使用多个标记 示例："%-10d"
数字	最小字段宽度 如果该字段不能容纳待打印的数字或字符串，系统会使用更宽的字段 示例："%4d"
.数字	精度 对于%e、%E 和%f 转换，表示小数点右边数字的位数 对于%g 和%G 转换，表示有效数字最大位数 对于%s 转换，表示待打印字符的最大数量 对于整型转换，表示待打印数字的最小位数 如有必要，使用前导 0 来达到这个位数 只使用.表示其后跟随一个 0，所以%.f 和%.0f 相同 示例："%5.2f"打印一个浮点数，字段宽度为 5 字符，其中小数点后有两位数字
h	和整型转换说明一起使用，表示 short int 或 unsigned short int 类型的值 示例："%hu"、"%hx"、"%6.4hd"
hh	和整型转换说明一起使用，表示 signed char 或 unsigned char 类型的值 示例："%hhu"、"%hhx"、"%6.4hhd"
j	和整型转换说明一起使用，表示 intmax_t 或 uintmax_t 类型的值。这些类型定义在 stdint.h 中 示例："%jd"、"%8jx"
l	和整型转换说明一起使用，表示 long int 或 unsigned long int 类型的值 示例："%ld"、"%8lu"
ll	和整型转换说明一起使用，表示 long long int 或 unsigned long long int 类型的值（C99） 示例："%lld"、"%8llu"
L	和浮点转换说明一起使用，表示 long double 类型的值 示例："%Ld"、"%10.4Le"
t	和整型转换说明一起使用，表示 ptrdiff_t 类型的值。ptrdiff_t 是两个指针差值的类型（C99） 示例："%td"、"%12ti"
z	和整型转换说明一起使用，表示 size_t 类型的值。size_t 是 sizeof 返回的类型（C99） 示例："%zd"、"%12zd"

注意　类型可移植性

sizeof 运算符以字节为单位返回类型或值的大小。这应该是某种形式的整数，但是标准只规定了该值是无符号整数。在不同的实现中，它可以是 unsigned int、unsigned long 甚至是 unsigned

long long。因此，如果要用 printf()函数显示 sizeof 表达式，根据不同系统，可能使用%u、%lu 或%llu。这意味着要查找你当前系统的用法，如果把程序移植到不同的系统还要进行修改。鉴于此，C 提供了可移植性更好的类型。首先，stddef.h 头文件（在包含 stdio.h 头文件时已包含其中）把 size_t 定义成系统使用 sizeof 返回的类型，这被称为底层类型（*underlying type*）。其次，printf()使用 z 修饰符表示打印相应的类型。同样，C 还定义了 ptrdiff_t 类型和 t 修饰符来表示系统使用的两个地址差值的底层有符号整数类型。

注意　float 参数的转换

对于浮点类型，有用于 double 和 long double 类型的转换说明，却没有 float 类型的。这是因为在 K&R C 中，表达式或参数中的 float 类型值会被自动转换成 double 类型。一般而言，ANSI C 不会把 float 自动转换成 double。然而，为保护大量假设 float 类型的参数被自动转换成 double 的现有程序，printf()函数中所有 float 类型的参数（对未使用显式原型的所有 C 函数都有效）仍自动转换成 double 类型。因此，无论是 K&R C 还是 ANSI C，都没有显示 float 类型值专用的转换说明。

表 4.5　printf()中的标记

标记	含义
-	待打印项左对齐。即，从字段的左侧开始打印该项项 示例："%-20s"
+	有符号值若为正，则在值前面显示加号；若为负，则在值前面显示减号 示例："%+6.2f"
空格	有符号值若为正，则在值前面显示前导空格（不显示任何符号）；若为负，则在值前面显示减号 +标记覆盖一个空格 示例："%6.2f"
#	把结果转换为另一种形式。如果是%o 格式，则以 0 开始；如果是%x 或%X 格式，则以 0x 或 0X 开始；对于所有的浮点格式，#保证了即使后面没有任何数字，也打印一个小数点字符。对于%g 和%G 格式，#防止结果后面的 0 被删除 示例："%#o"、"%#8.0f"、"%+#10.3e"
0	对于数值格式，用前导 0 代替空格填充字段宽度。对于整数格式，如果出现-标记或指定精度，则忽略该标记

1. 使用修饰符和标记的示例

接下来，用程序示例演示如何使用这些修饰符和标记。先来看看字段宽度在打印整数时的效果。考虑程序清单 4.7 中的程序。

程序清单 4.7　width.c 程序

```
/* width.c -- 字段宽度 */
#include <stdio.h>
#define PAGES 959
int main(void)
{
    printf("*%d*\n", PAGES);
    printf("*%2d*\n", PAGES);
     printf("*%10d*\n", PAGES);
```

```
    printf("*%-10d*\n", PAGES);

    return 0;
}
```

程序清单 4.7 通过 4 种不同的转换说明把相同的值打印了 4 次。程序中使用星号（*）标出每个字段的开始和结束。其输出结果如下所示：

```
*959*
*959*
*       959*
*959       *
```

第 1 个转换说明%d 不带任何修饰符，其对应的输出结果与带整数字段宽度的转换说明的输出结果相同。在默认情况下，没有任何修饰符的转换说明，就是这样的打印结果。第 2 个转换说明是%2d，其对应的输出结果应该是 2 字段宽度。因为待打印的整数有 3 位数字，所以字段宽度自动扩大以符合整数的长度。第 3 个转换说明是%10d，其对应的输出结果有 10 个空格宽度，实际上在两个星号之间有 7 个空格和 3 位数字，并且数字位于字段的右侧。最后一个转换说明是%-10d，其对应的输出结果同样是 10 个空格宽度，-标记说明打印的数字位于字段的左侧。熟悉它们的用法后，能很好地控制输出格式。试着改变 PAGES 的值，看看编译器如何打印不同位数的数字。

接下来看看浮点型格式。请输入、编译并运行程序清单 4.8 中的程序。

程序清单 4.8 floats.c 程序

```
// floats.c -- 一些浮点型修饰符的组合
#include <stdio.h>

int main(void)
{
    const double RENT = 3852.99;  // const 变量

    printf("*%f*\n", RENT);
    printf("*%e*\n", RENT);
    printf("*%4.2f*\n", RENT);
    printf("*%3.1f*\n", RENT);
    printf("*%10.3f*\n", RENT);
    printf("*%10.3E*\n", RENT);
    printf("*%+4.2f*\n", RENT);
    printf("*%010.2f*\n", RENT);

    return 0;
}
```

该程序中使用了 const 关键字，限定变量为只读。该程序的输出如下：

```
*3852.990000*
*3.852990e+03*
*3852.99*
*3853.0*
*  3852.990*
* 3.853E+03*
*+3852.99*
*0003852.99*
```

本例的第 1 个转换说明是%f。在这种情况下，字段宽度和小数点后面的位数均为系统默认设置，即字段宽度是容纳带打印数字所需的位数和小数点后打印 6 位数字。

第 2 个转换说明是%e。默认情况下，编译器在小数点的左侧打印 1 个数字，在小数点的右侧打印 6 个数字。这样打印的数字太多！解决方案是指定小数点右侧显示的位数，程序中接下来的 4 个例子就是这样做的。请注意，第 4 个和第 6 个例子对输出结果进行了四舍五入。另外，第 6 个例子用 E 代替了 e。

第 7 个转换说明中包含了+标记，这使得打印的值前面多了一个代数符号（+）。0 标记使得打印的值前面以 0 填充以满足字段要求。注意，转换说明%010.2f 的第 1 个 0 是标记，句点（.）之前、标记之后的数字（本例为 10）是指定的字段宽度。

尝试修改 RENT 的值，看看编译器如何打印不同大小的值。程序清单 4.9 演示了其他组合。

程序清单 4.9 flags.c 程序

```
/* flags.c -- 演示一些格式标记 */
#include <stdio.h>
int main(void)
{
    printf("%x %X %#x\n", 31, 31, 31);
    printf("**%d**% d**% d**\n", 42, 42, -42);
    printf("**%5d**%5.3d**%05d**%05.3d**\n", 6, 6, 6, 6);

    return 0;
}
```

该程序的输出如下：

```
1f 1F 0x1f
**42** 42**-42**
**    6** 006**00006**  006**
```

第 1 行输出中，1f 是十六进制数，等于十进制数 31。第 1 行 printf()语句中，根据%x 打印出 1f，%F 打印出 1F，%#x 打印出 0x1f。

第 2 行输出演示了如何在转换说明中用空格在输出的正值前面生成前导空格，负值前面不产生前导空格。这样的输出结果比较美观，因为打印出来的正值和负值在相同字段宽度下的有效数字位数相同。

第 3 行输出演示了如何在整型格式中使用精度（%5.3d）生成足够的前导 0 以满足最小位数的要求（本例是 3）。然而，使用 0 标记会使得编译器用前导 0 填充满整个字段宽度。最后，如果 0 标记和精度一起出现，0 标记会被忽略。

下面来看看字符串格式的示例。考虑程序清单 4.10 中的程序。

程序清单 4.10 stringf.c 程序

```
/* stringf.c -- 字符串格式 */
#include <stdio.h>
#define BLURB "Authentic imitation!"
int main(void)
{
    printf("[%2s]\n", BLURB);
     printf("[%24s]\n", BLURB);
    printf("[%24.5s]\n", BLURB);
    printf("[%-24.5s]\n", BLURB);

    return 0;
```

```
}
```

该程序的输出如下：

```
[Authentic imitation!]
[    Authentic imitation!]
[                    Authe]
[Authe                    ]
```

注意，虽然第 1 个转换说明是%2s，但是字段被扩大为可容纳字符串中的所有字符。还需注意，精度限制了待打印字符的个数。.5 告诉 printf()只打印 5 个字符。另外，-标记使得文本左对齐输出。

2. 学以致用

学习完以上几个示例，试试如何用一个语句打印以下格式的内容：

```
The NAME family just may be $XXX.XX dollars richer!
```

这里，NAME 和 XXX.XX 代表程序中变量（如 name[40]和 cash）的值。可参考以下代码：

```
printf("The %s family just may be $%.2f richer!\n",name,cash);
```

4.4.4 转换说明的意义

下面深入探讨一下转换说明的意义。转换说明把以二进制格式储存在计算机中的值转换成一系列字符（字符串）以便于显示。例如，数字 76 在计算机内部的存储格式是二进制数 01001100。%d 转换说明将其转换成字符 7 和 6，并显示为 76；%x 转换说明把相同的值（01001100）转换成十六进制记数法 4c；%c 转换说明把 01001100 转换成字符 L。

转换（*conversion*）可能 会误导读者认为原始值被转替换成转换后的值。实际上，转换说明是翻译说明，%d 的意思是“把给定的值翻译成十进制整数文本并打印出来”。

1. 转换不匹配

前面强调过，转换说明应该与待打印值的类型相匹配。通常都有多种选择。例如，如果要打印一个 int 类型的值，可以使用%d、%x 或%o。这些转换说明都可用于打印 int 类型的值，其区别在于它们分别表示一个值的形式不同。类似地，打印 double 类型的值时，可使用%f、%e 或%g。

转换说明与待打印值的类型不匹配会怎样？上一章中介绍过不匹配导致的一些问题。匹配非常重要，一定要牢记于心。程序清单 4.11 演示了一些不匹配的整型转换示例。

程序清单 4.11 intconv.c 程序

```
/* intconv.c -- 一些不匹配的整型转换 */
#include <stdio.h>
#define PAGES 336
#define WORDS 65618
int main(void)
{
    short num = PAGES;
    short mnum = -PAGES;

    printf("num as short and unsigned short:  %hd %hu\n", num,num);
    printf("-num as short and unsigned short: %hd %hu\n", mnum,mnum);
    printf("num as int and char: %d %c\n", num, num);
    printf("WORDS as int, short, and char: %d %hd %c\n",WORDS,WORDS, WORDS);

    return 0;
```

```
}
```

在我们的系统中，该程序的输出如下：

```
num as short and unsigned short: 336 336
-num as short and unsigned short: -336 65200
num as int and char: 336 P
WORDS as int, short, and char: 65618 82 R
```

请看输出的第 1 行，num 变量对应的转换说明%hd 和%hu 输出的结果都是 336。这没有任何问题。然而，第 2 行 mnum 变量对应的转换说明%u（无符号）输出的结果却为 65200，并非期望的 336。这是由于有符号 short int 类型的值在我们的参考系统中的表示方式所致。首先，short int 的大小是 2 字节；其次，系统使用**二进制补码**来表示有符号整数。这种方法，数字 0～32767 代表它们本身，而数字 32768～65535 则表示负数。其中，65535 表示-1，65534 表示-2，以此类推。因此，-336 表示为 65200（即，65536-336）。所以被解释成有符号 int 时，65200 代表-336；而被解释成无符号 int 时，65200 则代表 65200。一定要谨慎！一个数字可以被解释成两个不同的值。尽管并非所有的系统都使用这种方法来表示负整数，但要注意一点：别期望用%u 转换说明能把数字和符号分开。

第 3 行演示了如果把一个大于 255 的值转换成字符会发生什么情况。在我们的系统中，short int 是 2 字节，char 是 1 字节。当 printf()使用%c 打印 336 时，它只会查看储存 336 的 2 字节中的后 1 字节。这种截断（见图 4.8）相当于用一个整数除以 256，只保留其余数。在这种情况下，余数是 80，对应的 ASCII 值是字符 P。用专业术语来说，该数字被解释成“以 256 为模”（*modulo 256*），即该数字除以 256 后取其余数。

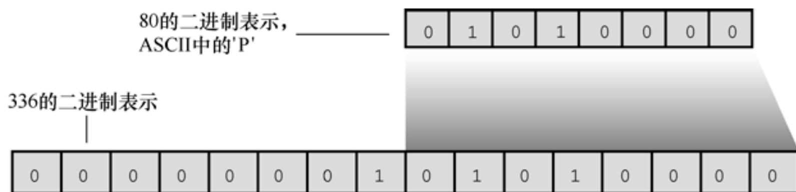


图 4.8　把 336 转换成字符

最后，我们在该系统中打印比 short int 类型最大整数（32767）更大的整数（65618）。这次，计算机也进行了求模运算。在本系统中，应把数字 65618 储存为 4 字节的 int 类型值。用%hd 转换说明打印时，printf()只使用最后 2 个字节。这相当于 65618 除以 65536 的余数。这里，余数是 82。鉴于负数的储存方法，如果余数在 32767～65536 范围内会被打印成负数。对于整数大小不同的系统，相应的处理行为类似，但是产生的值可能不同。

混淆整型和浮点型，结果更奇怪。考虑程序清单 4.12。

程序清单 4.12　floatcnv.c 程序

```
/* floatcnv.c -- 不匹配的浮点型转换 */
#include <stdio.h>
int main(void)
{
     float n1 = 3.0;
    double n2 = 3.0;
    long n3 = 2000000000;
    long n4 = 1234567890;

    printf("%.1e %.1e %.1e %.1e\n", n1, n2, n3, n4);
    printf("%ld %ld\n", n3, n4);
```

```
    printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);

    return 0;
}
```

在我们的系统中，该程序的输出如下：

```
3.0e+00 3.0e+00 3.1e+46 1.7e+266
2000000000 1234567890
0 1074266112 0 1074266112
```

第 1 行输出显示，%e 转换说明没有把整数转换成浮点数。考虑一下，如果使用%e 转换说明打印 n3（long 类型）会发生什么情况。首先，%e 转换说明让 printf()函数认为待打印的值是 double 类型（本系统中 double 为 8 字节）。当 printf()查看 n3（本系统中是 4 字节的值）时，除了查看 n3 的 4 字节外，还会查看查看 n3 相邻的 4 字节，共 8 字节单元。接着，它将 8 字节单元中的位组合解释成浮点数（如，把一部分位组合解释成指数）。因此，即使 n3 的位数正确，根据%e 转换说明和%ld 转换说明解释出来的值也不同。最终得到的结果是无意义的值。

第 1 行也说明了前面提到的内容：float 类型的值作为 printf()参数时会被转换成 double 类型。在本系统中，float 是 4 字节，但是为了 printf()能正确地显示该值，n1 被扩成 8 字节。

第 2 行输出显示，只要使用正确的转换说明，printf()就可以打印 n3 和 n4。

第 3 行输出显示，如果 printf()语句有其他不匹配的地方，即使用对了转换说明也会生成虚假的结果。用%ld 转换说明打印浮点数会失败，但是在这里，用%ld 打印 long 类型的数竟然也失败了！问题出在 C 如何把信息传递给函数。具体情况因编译器实现而异。“参数传递”框中针对一个有代表性的系统进行了讨论。

参数传递

参数传递机制因实现而异。下面以我们的系统为例，分析参数传递的原理。函数调用如下:

```
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

该调用告诉计算机把变量 n1、n2、、n3 和 n4 的值传递给程序。这是一种常见的参数传递方式。程序把传入的值放入被称为栈（*stack*）的内存区域。计算机根据变量类型（不是根据转换说明）把这些值放入栈中。因此，n1 被储存在栈中，占 8 字节（float 类型被转换成 double 类型）。同样，n2 也在栈中占 8 字节，而 n3 和 n4 在栈中分别占 4 字节。然后，控制转到 printf()函数。该函数根据转换说明（不是根据变量类型）从栈中读取值。%ld 转换说明表明 printf()应该读取 4 字节，所以 printf()读取栈中的前 4 字节作为第 1 个值。这是 n1 的前半部分，将被解释成一个 long 类型的整数。根据下一个%ld 转换说明，printf()再读取 4 字节，这是 n1 的后半部分，将被解释成第 2 个 long 类型的整数（见图 4.9）。类似地，根据第 3 个和第 4 个%ld，printf()读取 n2 的前半部分和后半部分，并解释成两个 long 类型的整数。因此，对于 n3 和 n4，虽然用对了转换说明，但 printf()还是读错了字节。

```
float n1; /* 作为 double 类型传递 */
double n2;
long n3, n4;
...
printf("%ld %ld %ld %ld\n", n1, n2, n3, n4);
```

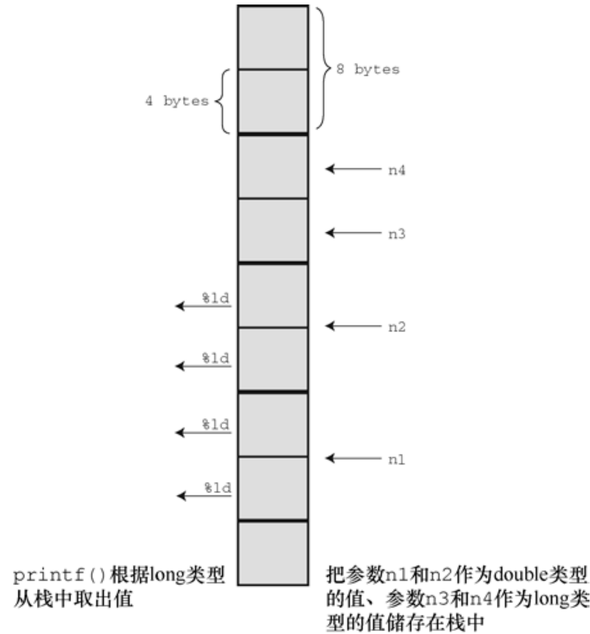


图 4.9　传递参数

2. printf()的返回值

第 2 章提到过，大部分 C 函数都有一个返回值，这是函数计算并返回给主调程序（*calling program*）的值。例如，C 库包含一个 sqrt()函数，接受一个数作为参数，并返回该数的平方根。可以把返回值赋给变量，也可以用于计算，还可以作为参数传递。总之，可以把返回值像其他值一样使用。printf()函数也有一个返回值，它返回打印字符的个数。如果有输出错误，printf()则返回一个负值（printf()的旧版本会返回不同的值）。

printf()的返回值是其打印输出功能的附带用途，通常很少用到，但在检查输出错误时可能会用到（如，在写入文件时很常用）。如果一张已满的 CD 或 DVD 拒绝写入时，程序应该采取相应的行动，例如终端蜂鸣 30 秒。不过，要实现这种情况必须先了解 if 语句。程序清单 4.13 演示了如何确定函数的返回值。

程序清单 4.13　prntval.c 程序

```
/* prntval.c -- printf()的返回值 */
#include <stdio.h>
int main(void)
{
    int bph2o = 212;
    int rv;

    rv = printf("%d F is water's boiling point.\n", bph2o);
    printf("The printf() function printed %d characters.\n",
            rv);
    return 0;
}
```

该程序的输出如下：

```
212 F is water's boiling point.
The printf() function printed 32 characters.
```

首先，程序用 rv = printf(...);的形式把 printf()的返回值赋给 rv。因此，该语句执行了两项任务：打印信息和给变量赋值。其次，注意计算针对所有字符数，包括空格和不可见的换行符（\n）。

3. 打印较长的字符串

有时，printf()语句太长，在屏幕上不方便阅读。如果空白（空格、制表符、换行符）仅用于分隔不同的部分，C 编译器会忽略它们。因此，一条语句可以写成多行，只需在不同部分之间输入空白即可。例如，程序清单 4.13 中的一条 printf()语句：

```
printf("The printf() function printed %d characters.\n",
        rv);
```

该语句在逗号和 rv 之间断行。为了让读者知道该行未完，示例缩进了 rv。C 编译器会忽略多余的空白。

但是，不能在双引号括起来的字符串中间断行。如果这样写：

```
printf("The printf() function printed %d
         characters.\n", rv);
```

C 编译器会报错：字符串常量中有非法字符。在字符串中，可以使用\n 来表示换行字符，但是不能通过按下 Enter（或 Return）键产生实际的换行符。

给字符串断行有 3 种方法，如程序清单 4.14 所示。

程序清单 4.14　longstrg.c 程序

```
/* longstrg.c --打印较长的字符串 */
#include <stdio.h>
int main(void)
{
    printf("Here's one way to print a ");
    printf("long string.\n");
    printf("Here's another way to print a \
long string.\n");
    printf("Here's the newest way to print a "
            "long string.\n");     /* ANSI C */

    return 0;
}
```

该程序的输出如下：

```
Here's one way to print a long string.
Here's another way to print a long string.
Here's the newest way to print a long string.
```

方法 1：使用多个 printf()语句。因为第 1 个字符串没有以\n 字符结束，所以第 2 个字符串紧跟第 1 个字符串末尾输出。

方法 2：用反斜杠（\）和 Enter（或 Return）键组合来断行。这使得光标移至下一行，而且字符串中不会包含换行符。其效果是在下一行继续输出。但是，下一行代码必须和程序清单中的代码一样从最左边开始。如果缩进该行，比如缩进 5 个空格，那么这 5 个空格就会成为字符串的一部分。

方法 3：ANSI C 引入的字符串连接。在两个用双引号括起来的字符串之间用空白隔开，C 编译器会把

多个字符串看作是一个字符串。因此，以下 3 种形式是等效的：

```
printf("Hello, young lovers, wherever you are.");
printf("Hello, young "     "lovers" ", wherever you are.");
printf("Hello, young lovers"
       ", wherever you are.");
```

上述方法中，要记得在字符串中包含所需的空格。如，"young""lovers"会成为"younglovers"，而"young " "lovers"才是"young lovers"。

4.4.5　使用 scanf()

刚学完输出，接下来我们转至输入——学习 scanf()函数。C 库包含了多个输入函数，scanf()是最通用的一个，因为它可以读取不同格式的数据。当然，从键盘输入的都是文本，因为键盘只能生成文本字符：字母、数字和标点符号。如果要输入整数 2014，就要键入字符 2、0、1、4。如果要将其储存为数值而不是字符串，程序就必须把字符依次转换成数值，这就是 scanf()要做的。scanf()把输入的字符串转换成整数、浮点数、字符或字符串，而 printf()正好与它相反，把整数、浮点数、字符和字符串转换成显示在屏幕上的文本。

scanf()和 printf()类似，也使用格式字符串和参数列表。scanf()中的格式字符串表明字符输入流的目标数据类型。两个函数主要的区别在参数列表中。printf()函数使用变量、常量和表达式，而 scanf()函数使用指向变量的指针。这里，读者不必了解如何使用指针，只需记住以下两条简单的规则：

- 如果用 scanf()读取基本变量类型的值，在变量名前加上一个&；
- 如果用 scanf()把字符串读入字符数组中，不要使用&。

程序清单 4.15 中的小程序演示了这两条规则。

程序清单 4.15　input.c 程序

```
// input.c -- 何时使用&
#include <stdio.h>
int main(void)
{
     int age;           // 变量
     float assets;      // 变量
     char pet[30];      // 字符数组，用于储存字符串

     printf("Enter your age, assets, and favorite pet.\n");
     scanf("%d %f", &age, &assets); // 这里要使用&
     scanf("%s", pet);              // 字符数组不使用&
     printf("%d $%.2f %s\n", age, assets, pet);

     return 0;
}
```

下面是该程序与用户交互的示例：

```
Enter your age, assets, and favorite pet.
38
92360.88 llama
38 $92360.88 llama
```

scanf()函数使用空白（换行符、制表符和空格）把输入分成多个字段。在依次把转换说明和字段匹

配时跳过空白。注意，上面示例的输入项（粗体部分是用户的输入）分成了两行。只要在每个输入项之间输入至少一个换行符、空格或制表符即可，可以在一行或多行输入：

```
Enter your age, assets, and favorite pet.
 42

    2121.45

    guppy
42 $2121.45 guppy
```

唯一例外的是%c 转换说明。根据%c，scanf()会读取每个字符，包括空白。我们稍后详述这部分。

scanf()函数所用的转换说明与 printf()函数几乎相同。主要的区别是，对于 float 类型和 double 类型，printf()都使用%f、%e、%E、%g 和%G 转换说明。而 scanf()只把它们用于 float 类型，对于 double 类型时要使用 l 修饰符。表 4.6 列出了 C99 标准中常用的转换说明。

表 4.6　ANSI C 中 scanf()的转换说明

转换说明	含义
%c	把输入解释成字符
%d	把输入解释成有符号十进制整数
%e、%f、%g、%a	把输入解释成浮点数（C99 标准新增了%a）
%E、%F、%G、%A	把输入解释成浮点数（C99 标准新增了%A）
%i	把输入解释成有符号十进制整数
%o	把输入解释成有符号八进制整数
%p	把输入解释成指针（地址）
%s	把输入解释成字符串。从第 1 个非空白字符开始，到下一个空白字符之前的所有字符都是输入
%u	把输入解释成无符号十进制整数
%x、%X	把输入解释成有符号十六进制整数

可以在表 4.6 所列的转换说明中（百分号和转换字符之间）使用修饰符。如果要使用多个修饰符，必须按表 4.7 所列的顺序书写。

表 4.7　scanf()转换说明中的修饰符

转换说明	含义
*	抑制赋值（详见后面解释） 示例："%*d"
数字	最大字段宽度。输入达到最大字段宽度处，或第 1 次遇到空白字符时停止 示例："%10s"
hh	把整数作为 signed char 或 unsigned char 类型读取 示例："%hhd"、"%hhu"
ll	把整数作为 long long 或 unsigned long long 类型读取（C99） 示例："%lld"、"%llu"

续表

转换说明	含义
h、l 或 L	"%hd"和"%hi"表明把对应的值储存为 short int 类型 "%ho"、"%hx"和"%hu"表明把对应的值储存为 unsigned shot int 类型 "%ld"和"%li"表明把对应的值储存为 long 类型 "%lo"、"%lx"和"%lu"表明把对应的值储存为 unsigned long 类型 "%le"、"%lf"和"%lg"表明把对应的值储存为 double 类型 在 e、f 和 g 前面使用 L 而不是 l，表明把对应的值被储存为 long double 类型。如果没有修饰符，d、i、o 和 x 表明对应的值被储存为 int 类型，f 和 g 表明把对应的值储存为 float 类型
j	在整型转换说明后面时，表明使用 intmax_t 或 uintmax_t 类型（C99） 示例："%zd"、"%zo"
z	在整型转换说明后面时，表明使用 sizeof 的返回类型（C99）
t	在整型转换说明后面时，表明使用表示两个指针差值的类型（C99） 示例："%td"、"%tx"

如你所见，使用转换说明比较复杂，而且这些表中还省略了一些特性。省略的主要特性是，从高度格式化源中读取选定数据，如穿孔卡或其他数据记录。因为在本书中，scanf()主要作为与程序交互的便利工具，所以我们不在书中讨论更复杂的特性。

1. 从 scanf()角度看输入

接下来，我们更详细地研究 scanf()怎样读取输入。假设 scanf()根据一个%d 转换说明读取一个整数。scanf()函数每次读取一个字符，跳过所有的空白字符，直至遇到第 1 个非空白字符才开始读取。因为要读取整数，所以 scanf()希望发现一个数字字符或者一个符号（+或-）。如果找到一个数字或符号，它便保存该字符，并读取下一个字符。如果下一个字符是数字，它便保存该数字并读取下一个字符。scanf()不断地读取和保存字符，直至遇到非数字字符。如果遇到一个非数字字符，它便认为读到了整数的末尾。然后，scanf()把非数字字符放回输入。这意味着程序在下一次读取输入时，首先读到的是上一次读取丢弃的非数字字符。最后，scanf()计算已读取数字（可能还有符号）相应的数值，并将计算后的值放入指定的变量中。

如果使用字段宽度，scanf()会在字段结尾或第 1 个空白字符处停止读取(满足两个条件之一便停止)。

如果第 1 个非空白字符是 A 而不是数字，会发生什么情况？scanf()将停在那里，并把 A 放回输入中，不会把值赋给指定变量。程序在下一次读取输入时，首先读到的字符是 A。如果程序只使用%d 转换说明，scanf()就一直无法越过 A 读下一个字符。另外，如果使用带多个转换说明的 scanf()，C 规定在第 1 个出错处停止读取输入。

用其他数值匹配的转换说明读取输入和用%d 的情况相同。区别在于 scanf()会把更多字符识别成数字的一部分。例如，%x 转换说明要求 scanf()识别十六进制数 a～f 和 A～F。浮点转换说明要求 scanf()识别小数点、e 记数法（指数记数法）和新增的 p 记数法（十六进制指数记数法）。

如果使用%s 转换说明，scanf()会读取除空白以外的所有字符。scanf()跳过空白开始读取第 1 个非空白字符，并保存非空白字符直到再次遇到空白。这意味着 scanf()根据%s 转换说明读取一个单词，即不包含空白字符的字符串。如果使用字段宽度，scanf()在字段末尾或第 1 个空白字符处停止读取。无法利用字段宽度让只有一个%s 的 scanf()读取多个单词。最后要注意一点：当 scanf()把字符串放进指定数组中时，它会在字符序列的末尾加上'\0'，让数组中的内容成为一个 C 字符串。

实际上，在 C 语言中 scanf()并不是最常用的输入函数。这里重点介绍它是因为它能读取不同类型的

数据。C 语言还有其他的输入函数，如 getchar()和 fgets()。这两个函数更适合处理一些特殊情况，如读取单个字符或包含空格的字符串。我们将在第 7 章、第 11 章、第 13 章中讨论这些函数。目前，无论程序中需要读取整数、小数、字符还是字符串，都可以使用 scanf()函数。

2. 格式字符串中的普通字符

scanf()函数允许把普通字符放在格式字符串中。除空格字符外的普通字符必须与输入字符串严格匹配。例如，假设在两个转换说明中添加一个逗号：

```
scanf("%d,%d", &n, &m);
```

scanf()函数将其解释成：用户将输入一个数字、一个逗号，然后再输入一个数字。也就是说，用户必须像下面这样进行输入两个整数：

```
88,121
```

由于格式字符串中，%d 后面紧跟逗号，所以必须在输入 88 后再输入一个逗号。但是，由于 scanf()会跳过整数前面的空白，所以下面两种输入方式都可以：

```
88, 121
```

和

```
88,
121
```

格式字符串中的空白意味着跳过下一个输入项前面的所有空白。例如，对于下面的语句：

```
scanf("%d ,%d", &n, &m);
```

以下的输入格式都没问题：

```
88,121
88 ,121
88 , 121
```

请注意，“所有空白”的概念包括没有空格的特殊情况。

除了%c，其他转换说明都会自动跳过待输入值前面所有的空白。因此，scanf("%d%d", &n, &m)与 scanf("%d %d", &n, &m)的行为相同。对于%c，在格式字符串中添加一个空格字符会有所不同。例如，如果把%c 放在格式字符串中的空格前面，scanf()便会跳过空格，从第 1 个非空白字符开始读取。也就是说，scanf("%c", &ch)从输入中的第 1 个字符开始读取，而 scanf(" %c", &ch)则从第 1 个非空白字符开始读取。

3. scanf()的返回值

scanf()函数返回成功读取的项数。如果没有读取任何项，且需要读取一个数字而用户却输入一个非数值字符串，scanf()便返回 0。当 scanf()检测到“文件结尾”时，会返回 EOF（EOF 是 stdio.h 中定义的特殊值，通常用#define 指令把 EOF 定义为-1）。我们将在第 6 章中讨论文件结尾的相关内容以及如何利用 scanf()的返回值。在读者学会 if 语句和 while 语句后，便可使用 scanf()的返回值来检测和处理不匹配的输入。

4.4.6 printf()和 scanf()的*修饰符

printf()和 scanf()都可以使用*修饰符来修改转换说明的含义。但是，它们的用法不太一样。首先，我们来看 printf()的*修饰符。

如果你不想预先指定字段宽度，希望通过程序来指定，那么可以用*修饰符代替字段宽度。但还是要用一个参数告诉函数，字段宽度应该是多少。也就是说，如果转换说明是%*d，那么参数列表中应包含*和 d 对应的值。这个技巧也可用于浮点值指定精度和字段宽度。程序清单 4.16 演示了相关用法。

程序清单 4.16　varwid.c 程序

```
/* varwid.c -- 使用变宽输出字段 */
#include <stdio.h>
int main(void)
{
    unsigned width, precision;
    int number = 256;
    double weight = 242.5;

    printf("Enter a field width:\n");
    scanf("%d", &width);
    printf("The number is :%*d:\n", width, number);
    printf("Now enter a width and a precision:\n");
    scanf("%d %d", &width, &precision);
    printf("Weight = %*.*f\n", width, precision, weight);
    printf("Done!\n");

    return 0;
}
```

变量 width 提供字段宽度，number 是待打印的数字。因为转换说明中*在 d 的前面，所以在 printf()的参数列表中，width 在 number 的前面。同样，width 和 precision 提供打印 weight 的格式化信息。下面是一个运行示例：

```
Enter a field width:
6
The number is : 256:
Now enter a width and a precision:
8 3
Weight = 242.500
Done!
```

这里，用户首先输入 6，因此 6 是程序使用的字段宽度。类似地，接下来用户输入 8 和 3，说明字段宽度是 8，小数点后面显示 3 位数字。一般而言，程序应根据 weight 的值来决定这些变量的值。

scanf()中*的用法与此不同。把*放在%和转换字符之间时，会使得 scanf()跳过相应的输出项。程序清单 4.17 就是一个例子。

程序清单 4.17　skip2.c 程序

```
/* skiptwo.c -- 跳过输入中的前两个整数 */
#include <stdio.h>
int main(void)
{
    int n;

    printf("Please enter three integers:\n");
    scanf("%*d %*d %d", &n);
    printf("The last integer was %d\n", n);

    return 0;
}
```

程序清单 4.17 中的 scanf()指示：跳过两个整数，把第 3 个整数拷贝给 n。下面是一个运行示例：

```
Please enter three integers:
```

```
2013 2014 2015
The last integer was 2015
```

在程序需要读取文件中特定列的内容时，这项跳过功能很有用。

4.4.7 printf()的用法提示

想把数据打印成列，指定固定字段宽度很有用。因为默认的字段宽度是待打印数字的宽度，如果同一列中打印的数字位数不同，那么下面的语句：

```
printf("%d %d %d\n", val1, val2, val3);
```

打印出来的数字可能参差不齐。例如，假设执行 3 次 printf()语句，用户输入不同的变量，其输出可能是这样：

```
12 234 1222
4 5 23
22334 2322 10001
```

使用足够大的固定字段宽度可以让输出整齐美观。例如，若使用下面的语句：

```
printf("%9d %9d %9d\n", val1, val2, val3);
```

上面的输出将变成：

```
      12       234      1222
       4         5        23
   22334      2322     10001
```

在两个转换说明中间插入一个空白字符，可以确保即使一个数字溢出了自己的字段，下一个数字也不会紧跟该数字一起输出（这样两个数字看起来像是一个数字）。这是因为格式字符串中的普通字符（包括空格）会被打印出来。

另一方面，如果要在文字中嵌入一个数字，通常指定一个小于或等于该数字宽度的字段会比较方便。这样，输出数字的宽度正合适，没有不必要的空白。例如，下面的语句：

```
printf("Count Beppo ran %.2f miles in 3 hours.\n", distance);
```

其输出如下：

```
Count Beppo ran 10.22 miles in 3 hours.
```

如果把转换说明改为%10.2f，则输出如下：

```
Count Beppo ran      10.22 miles in 3 hours.
```

本地化设置

美国和世界上的许多地区都使用一个点来分隔十进制值的整数部分和小数部分，如 3.14159。然而，许多其他地区用逗号来分隔，如 3,14159。读者可能注意到了，printf()和 scanf()都没有提供逗号的转换说明。C 语言考虑了这种情况。本书附录 B 的参考资料 V 中介绍了 C 支持的本地化概念，因此 C 程序可以选择特定的本地化设置。例如，如果指定了荷兰语言环境，printf()和 scanf()在显示和读取浮点值时会使用本地惯例（在这种情况下，用逗号代替点分隔浮点值的整数部分和小数部分）。另外，一旦指定了环境，便可在代码的数字中使用逗号：

```
double pi = 3,14159; // 荷兰本地化设置
```

C 标准有两个本地化设置："C"和""（空字符串）。默认情况下，程序使用"C"本地化设置，基本上符合美国的用法习惯。而""本地化设置可以替换当前系统中使用的本地语言环境。原则上，这与"C"本地化设置相同。事实上，大部分操作系统（如 UNIX、Linux 和 Windows）都提供本地化设置选项列表，只不过它们提供的列表可能不同。

4.5 关键概念

C 语言用 char 类型表示单个字符，用字符串表示字符序列。字符常量是一种字符串形式，即用双引号把字符括起来："Good luck, my friend"。可以把字符串储存在字符数组（由内存中相邻的字节组成）中。字符串，无论是表示成字符常量还是储存在字符数组中，都以一个叫做**空字符**的隐藏字符结尾。

在程序中，最好用#define 定义数值常量，用 const 关键字声明的变量为只读变量。在程序中使用符号常量（明示常量），提高了程序的可读性和可维护性。

C 语言的标准输入函数（scanf()）和标准输出函数（printf()）都使用一种系统。在该系统中，第 1 个参数中的转换说明必须与后续参数中的值相匹配。例如，int 转换说明%d 与一个浮点值匹配会产生奇怪的结果。必须格外小心，确保转换说明的数量和类型与函数的其余参数相匹配。对于 scanf()，一定要记得在变量名前加上地址运算符（&）。

空白字符（制表符、空格和换行符）在 scanf()处理输入时起着至关重要的作用。除了%c 模式（读取下一个字符），scanf()在读取输入时会跳过非空白字符前的所有空白字符，然后一直读取字符，直至遇到空白字符或与正在读取字符不匹配的字符。考虑一下，如果 scanf()根据不同的转换说明读取相同的输入行，会发生什么情况。假设有如下输入行：

```
-13.45e12# 0
```

如果其对应的转换说明是%d，scanf()会读取 3 个字符（-13）并停在小数点处，小数点将被留在输入中作为下一次输入的首字符。如果其对应的转换说明是%f，scanf()会读取-13.45e12，并停在#符号处，而#将被留在输入中作为下一次输入的首字符；然后，scanf()把读取的字符序列-13.45e12 转换成相应的浮点值，并储存在float类型的目标变量中。如果其对应的转换说明是%s，scanf()会读取-13.45e12#，并停在空格处，空格将被留在输入中作为下一次输入的首字符；然后，scanf()把这 10 个字符的字符码储存在目标字符数组中，并在末尾加上一个空字符。如果其对应的转换说明是%c，scanf()只会读取并储存第 1 个字符，该例中是一个空格[1]。

4.6 本章小结

字符串是一系列被视为一个处理单元的字符。在 C 语言中，字符串是以空字符（ASCII 码是 0）结尾的一系列字符。可以把字符串储存在字符数组中。数组是一系列同类型的项或元素。下面声明了一个名为 name、有 30 个 char 类型元素的数组：

```
char name[30];
```

要确保有足够多的元素来储存整个字符串（包括空字符）。

字符串常量是用双引号括起来的字符序列，如："This is an example of a string"。

scanf()函数（声明在 string.h 头文件中）可用于获得字符串的长度（末尾的空字符不计算在内）。scanf()函数中的转换说明是%s 时，可读取一个单词。

C 预处理器为预处理器指令（以#符号开始）查找源代码程序，并在开始编译程序之前处理它们。处理器根据#include 指令把另一个文件中的内容添加到该指令所在的位置。#define 指令可以创建明示常量（符号常量），即代表常量的符号。limits.h 和 float.h 头文件用#define 定义了一组表示整型和浮点型不同属性的符号常量。另外，还可以使用 const 限定符创建定义后就不能修改的变量。

[1] 注意，“ -13.45e12# 0 ”的负号前面有一个空格。——译者注

printf()和scanf()函数对输入和输出提供多种支持。两个函数都使用格式字符串，其中包含的转换说明表明待读取或待打印数据项的数量和类型。另外，可以使用转换说明控制输出的外观：字段宽度、小数位和字段内的布局。

4.7　复习题

复习题的参考答案在附录A中。

1. 再次运行程序清单 4.1，但是在要求输入名时，请输入名和姓（根据英文书写习惯，名和姓中间有一个空格），看看会发生什么情况？为什么？
2. 假设下列示例都是完整程序中的一部分，它们打印的结果分别是什么？

   a. `printf("He sold the painting for $%2.2f.\n", 2.345e2);`

   b. `printf("%c%c%c\n", 'H', 105, '\41');`

   c. `#define Q "His Hamlet was funny without being vulgar."`
      `printf("%s\nhas %d characters.\n", Q, strlen(Q));`

   d. `printf("Is %2.2e the same as %2.2f?\n", 1201.0, 1201.0);`
3. 在第2题的c中，要输出包含双引号的字符串Q，应如何修改？
4. 找出下面程序中的错误。

```
define B booboo
define X 10
main(int)
{
    int age;
    char name;
    printf("Please enter your first name.");
    scanf("%s", name);
    printf("All right, %c, what's your age?\n", name);
    scanf("%f", age);
    xp = age + X;
    printf("That's a %s! You must be at least %d.\n", B, xp);
    rerun 0;
}
```

5. 假设一个程序的开头是这样：

```
#define BOOK "War and Peace"
int main(void)
{
    float cost =12.99;
    float percent = 80.0;
```

   请构造一个使用BOOK、cost和percent的printf()语句，打印以下内容：

```
This copy of "War and Peace" sells for $12.99.
That is 80% of list.
```

6. 打印下列各项内容要分别使用什么转换说明？

   a. 一个字段宽度与位数相同的十进制整数

   b. 一个形如8A、字段宽度为4的十六进制整数

   c. 一个形如232.346、字段宽度为10的浮点数

d．一个形如 2.33e+002、字段宽度为 12 的浮点数

e．一个字段宽度为 30、左对齐的字符串

7．打印下面各项内容要分别使用什么转换说明？

a．字段宽度为 15 的 unsigned long 类型的整数

b．一个形如 0x8a、字段宽度为 4 的十六进制整数

c．一个形如 2.33E+02、字段宽度为 12、左对齐的浮点数

d．一个形如+232.346、字段宽度为 10 的浮点数

e．一个字段宽度为 8 的字符串的前 8 个字符

8．打印下面各项内容要分别使用什么转换说明？

a．一个字段宽度为 6、最少有 4 位数字的十进制整数

b．一个在参数列表中给定字段宽度的八进制整数

c．一个字段宽度为 2 的字符

d．一个形如+3.13、字段宽度等于数字中字符数的浮点数

e．一个字段宽度为 7、左对齐字符串中的前 5 个字符

9．分别写出读取下列各输入行的 scanf()语句，并声明语句中用到变量和数组。

a．101

b．22.32 8.34E 09

c．linguini

d．catch 22

e．catch 22 （但是跳过catch）

10．什么是空白？

11．下面的语句有什么问题？如何修正？

```
printf("The double type is %z bytes..\n", sizeof(double));
```

12．假设要在程序中用圆括号代替花括号，以下方法是否可行？

```
#define ( {
#define ) }
```

4.8　编程练习

1．编写一个程序，提示用户输入名和姓，然后以“名,姓”的格式打印出来。

2．编写一个程序，提示用户输入名和姓，并执行一下操作：

a．打印名和姓，包括双引号；

b．在宽度为 20 的字段右端打印名和姓，包括双引号；

c．在宽度为 20 的字段左端打印名和姓，包括双引号；

d．在比姓名宽度宽 3 的字段中打印名和姓。

3．编写一个程序，读取一个浮点数，首先以小数点记数法打印，然后以指数记数法打印。用下面的格式进行输出（系统不同，指数记数法显示的位数可能不同）：

a．输入 21.3 或 2.1e+001；

b. 输入+21.290或2.129E+001；

4. 编写一个程序，提示用户输入身高（单位：英寸）和姓名，然后以下面的格式显示用户刚输入的信息：

```
Dabney, you are 6.208 feet tall
```

使用 float 类型，并用/作为除号。如果你愿意，可以要求用户以厘米为单位输入身高，并以米为单位显示出来。

5. 编写一个程序，提示用户输入以兆位每秒（Mb/s）为单位的下载速度和以兆字节（MB）为单位的文件大小。程序中应计算文件的下载时间。注意，这里 1 字节等于 8 位。使用 float 类型，并用/作为除号。该程序要以下面的格式打印 3 个变量的值（下载速度、文件大小和下载时间），显示小数点后面两位数字：

```
At 18.12 megabits per second, a file of 2.20 megabytes
downloads in 0.97 seconds.
```

6. 编写一个程序，先提示用户输入名，然后提示用户输入姓。在一行打印用户输入的名和姓，下一行分别打印名和姓的字母数。字母数要与相应名和姓的结尾对齐，如下所示：

```
Melissa Honeybee
      7        8
```

接下来，再打印相同的信息，但是字母个数与相应名和姓的开头对齐，如下所示：

```
Melissa Honeybee
7       8
```

7. 编写一个程序，将一个 double 类型的变量设置为 1.0/3.0，一个 float 类型的变量设置为 1.0/3.0。分别显示两次计算的结果各 3 次：一次显示小数点后面 6 位数字；一次显示小数点后面 12 位数字；一次显示小数点后面 16 位数字。程序中要包含 float.h 头文件，并显示 FLT_DIG 和 DBL_DIG 的值。1.0/3.0 的值与这些值一致吗？

8. 编写一个程序，提示用户输入旅行的里程和消耗的汽油量。然后计算并显示消耗每加仑汽油行驶的英里数，显示小数点后面一位数字。接下来，使用 1 加仑大约 3.785 升，1 英里大约为 1.609 千米，把单位是英里/加仑的值转换为升/100 公里（欧洲通用的燃料消耗表示法），并显示结果，显示小数点后面 1 位数字。注意，美国采用的方案测量消耗单位燃料的行程（值越大越好），而欧洲则采用单位距离消耗的燃料测量方案（值越低越好）。使用#define 创建符号常量或使用 const 限定符创建变量来表示两个转换系数。

第5章 运算符、表达式和语句

本章介绍以下内容：

- 关键字：while、typedef
- 运算符：=、-、*、/、%、++、--、(类型名)
- C 语言的各种运算符，包括用于普通数学运算的运算符
- 运算符优先级以及语句、表达式的含义
- while 循环
- 复合语句、自动类型转换和强制类型转换
- 如何编写带有参数的函数

现在，读者已经熟悉了如何表示数据，接下来我们学习如何处理数据。C 语言为处理数据提供了大量的操作，可以在程序中进行算术运算、比较值的大小、修改变量、逻辑地组合关系等。我们先从基本的算术运算（加、减、乘、除）开始。

组织程序是处理数据的另一个方面，让程序按正确的顺序执行各个步骤。C 有许多语言特性，帮助你完成组织程序的任务。循环就是其中一个特性，本章中你将窥其大概。循环能重复执行行为，让程序更有趣、更强大。

5.1 循环简介

程序清单 5.1 是一个简单的程序示例，该程序进行了简单的运算，计算穿 9 码男鞋的脚长（单位：英寸）。为了让读者体会循环的好处，程序的第 1 个版本演示了不使用循环编程的局限性。

程序清单 5.1 shoes1.c **程序**

```
/* shoes1.c -- 把鞋码转换成英寸 */
#include <stdio.h>
#define ADJUST 7.31                  // 字符常量
int main(void)
{
    const double SCALE = 0.333;// const 变量
    double shoe, foot;

    shoe = 9.0;
    foot = SCALE * shoe + ADJUST;
    printf("Shoe size (men's)    foot length\n");
    printf("%10.1f %15.2f inches\n", shoe, foot);

    return 0;
}
```

该程序的输出如下：

```
Shoe size (men's) foot length
      9.0         10.31 inches
```

该程序演示了用#define 指令创建符号常量和用 const 限定符创建在程序运行过程中不可更改的变量。程序使用了乘法和加法，假定用户穿 9 码的鞋，以英寸为单位打印用户的脚长。你可能会说："这太简单了，我用笔算比敲程序还要快。"说得没错。写出来的程序只使用一次（本例即只根据一只鞋的尺码计算一次脚长），实在是浪费时间和精力。如果写成交互式程序会更有用，但是仍无法利用计算机的优势。

应该让计算机做一些重复计算的工作。毕竟，需要重复计算是使用计算机的主要原因。C 提供多种方法做重复计算，我们在这里简单介绍一种——while 循环。它能让你对运算符做更有趣地探索。程序清单 5.2 演示了用循环改进后的程序。

程序清单 5.2　shoes2.c 程序

```
/* shoes2.c -- 计算多个不同鞋码对应的脚长 */
#include <stdio.h>
#define ADJUST 7.31                  // 字符常量
int main(void)
{
    const double SCALE = 0.333;// const 变量
    double shoe, foot;

    printf("Shoe size (men's)    foot length\n");
    shoe = 3.0;
    while (shoe < 18.5)              /* while 循环开始 */
    {                                /* 块开始   */
        foot = SCALE * shoe + ADJUST;
        printf("%10.1f %15.2f inches\n", shoe, foot);
        shoe = shoe + 1.0;
    }                                /* 块结束       */
    printf("If the shoe fits, wear it.\n");

    return 0;
}
```

下面是 shoes2.c 程序的输出（...表示并未显示完整，有删节）：

```
Shoe size (men's) foot length
       3.0         8.31 inches
       4.0         8.64 inches
       5.0         8.97 inches
       6.0         9.31 inches
...
      16.0        12.64 inches
      17.0        12.97 inches
      18.0        13.30 inches
If the shoe fits, wear it.
```

（如果读者对此颇有研究，应该知道该程序不符合实际情况。程序中假定了一个统一的鞋码系统。）

下面解释一下 while 循环的原理。当程序第 1 次到达 while 循环时，会检查圆括号中的条件是否为真。该程序中，条件表达式如下：

```
shoe < 18.5
```

符号<的意思是小于。变量 shoe 被初始化为 3.0，显然小于 18.5。因此，该条件为真，程序进入块

中继续执行，把尺码转换成英寸。然后打印计算的结果。下一条语句把 shoe 增加 1.0，使 shoe 的值为 4.0：

```
shoe = shoe + 1.0;
```

此时，程序返回 while 入口部分检查条件。为何要返回 while 的入口部分？因为上面这条语句的下面是右花括号（}），代码使用一对花括号（{}）来标出 while 循环的范围。花括号之间的内容就是要被重复执行的内容。花括号以及被花括号括起来的部分被称为块（*block*）。现在，回到程序中。因为 4 小于 18.5，所以要重复执行被花括号括起来的所有内容（用计算机术语来说就是，程序循环这些语句）。该循环过程一直持续到 shoe 的值为 19.0。此时，由于 19.0 小于 18.5，所以该条件为假：

```
shoe < 18.5
```

出现这种情况后，控制转到紧跟 while 循环后面的第 1 条语句。该例中，是最后的 printf()语句。

可以很方便地修改该程序用于其他转换。例如，把 SCALE 设置成 1.8、ADJUST 设置成 32.0，该程序便可把摄氏温度转换成华氏温度；把 SCALE 设置成 0.6214、ADJUST 设置成 0，该程序便可把公里转换成英里。注意，修改了设置后，还要更改打印的消息，以免前后表述不一。

通过 while 循环能便捷灵活地控制程序。现在，我们来学习程序中会用到的基本运算符。

5.2 基本运算符

C 用运算符（*operator*）表示算术运算。例如，+运算符使在它两侧的值加在一起。如果你觉得术语“运算符”很奇怪，那么请记住东西总得有个名称。与其叫“那些东西”或“运算处理符”，还不如叫“运算符”。现在，我们介绍一下用于基本算术运算的运算符：=、+、-、*和/（C 没有指数运算符。不过，C 的标准数学库提供了一个 pow()函数用于指数运算。例如，pow(3.5, 2.2)返回 3.5 的 2.2 次幂）。

5.2.1 赋值运算符：=

在 C 语言中，=并不意味着“相等”，而是一个赋值运算符。下面的赋值表达式语句：

```
bmw = 2002;
```

把值 2002 赋给变量 bmw。也就是说，=号左侧是一个变量名，右侧是赋给该变量的值。符号=被称为赋值运算符。另外，上面的语句不读作“bmw 等于 2002”，而读作“把值 2002 赋给变量 bmw”。赋值行为从右往左进行。

也许变量名和变量值的区别看上去微乎其微，但是，考虑下面这条常用的语句：

```
i = i + 1;
```

对数学而言，这完全行不通。如果给一个有限的数加上 1，它不可能“等于”原来的数。但是，在计算机赋值表达式语句中，这很合理。该语句的意思是：找出变量 i 的值，把该值加 1，然后把新值赋值变量 i（见图 5.1）。

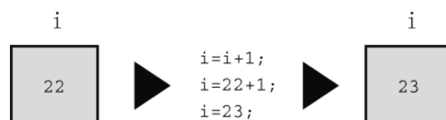


图 5.1 语句 i = i + 1;

在 C 语言中，类似这样的语句没有意义（实际上是无效的）：

```
2002 = bmw;
```

因为在这种情况下，2002 被称为右值（*rvale*），只能是字面常量。不能给常量赋值，常量本身就是它

的值。因此，在编写代码时要记住，=号左侧的项必须是一个变量名。实际上，赋值运算符左侧必须引用一个存储位置。最简单的方法就是使用变量名。不过，后面章节还会介绍“指针”，可用于指向一个存储位置。概括地说，C 使用*可修改的左值*（*modifiable lvalue*）标记那些可赋值的实体。也许“可修改的左值”不太好懂，我们再来看一些定义。

几个术语：数据对象、左值、右值和运算符

赋值表达式语句的目的是把值储存到内存位置上。用于储存值的数据存储区域统称为*数据对象*（*data object*）。C 标准只有在提到这个概念时才会用到*对象*这个术语。使用变量名是标识对象的一种方法。除此之外，还有其他方法，但是要在后面的章节中才学到。例如，可以指定数组的元素、结构的成员，或者使用指针表达式（指针中储存的是它所指向对象的地址）。*左值*（*lvalue*）是 C 语言的术语，用于标识特定数据对象的名称或表达式。因此，对象指的是实际的数据存储，而左值是用于标识或定位存储位置的标签。

对于早期的 C 语言，提到左值意味着：

1．它指定一个对象，所以引用内存中的地址；

2．它可用在赋值运算符的左侧，*左值*（*lvalue*）中的 l 源自 left。

但是后来，标准中新增了 `const` 限定符。用 `const` 创建的变量不可修改。因此，`const` 标识符满足上面的第 1 项，但是不满足第 2 项。一方面 C 继续把标识对象的表达式定义为左值，一方面某些左值却不能放在赋值运算符的左侧。有些左值不能用于赋值运算符的左侧。此时，标准对左值的定义已经不能满足当前的状况。

为此，C 标准新增了一个术语：*可修改的左值*（*modifiable lvalue*），用于标识可修改的对象。所以，赋值运算符的左侧应该是可修改的左值。当前标准建议，使用术语*对象定位值*（*object locator value*）更好。

右值（*rvalue*）指的是能赋值给可修改左值的量，且本身不是左值。例如，考虑下面的语句：

```
bmw = 2002;
```

这里，`bmw` 是可修改的左值，`2002` 是右值。读者也许猜到了，右值中的 r 源自 right。右值可以是常量、变量或其他可求值的表达式（如，函数调用）。实际上，当前标准在描述这一概念时使用的是*表达式的值*（*value of an expression*），而不是右值。

我们看几个简单的示例：

```
int ex;
int why;
int zee;
const int TWO = 2;
why = 42;
zee = why;
ex = TWO * (why + zee);
```

这里，`ex`、`why` 和 `zee` 都是可修改的左值（或对象定位值），它们可用于赋值运算符的左侧和右侧。`TWO` 是不可改变的左值，它只能用于赋值运算符的右侧（在该例中，`TWO` 被初始化为 `2`，这里的=运算符表示初始化而不是赋值，因此并未违反规则）。同时，`42` 是右值，它不能引用某指定内存位置。另外，`why` 和 `zee` 是可修改的左值，表达式`(why + zee)`是右值，该表达式不能表示特定内存位置，而且也不能给它赋值。它只是程序计算的一个临时值，在计算完毕后便会被丢弃。

在学习名称时，被称为“项”（如，赋值运算符左侧的项）的就是*运算对象*（*operand*）。运算对象是运算符操作的对象。例如，可以把吃汉堡描述为：“吃”运算符操作“汉堡”运算对象。类似地可以说，=运算符的左侧运算对象应该是可修改的左值。

C 的基本赋值运算符有些与众不同，请看程序清单 5.3。

程序清单 5.3　golf.c 程序

```
/* golf.c -- 高尔夫锦标赛记分卡 */
#include <stdio.h>
int main(void)
{
    int jane, tarzan, cheeta;

    cheeta = tarzan = jane = 68;
    printf("                cheeta   tarzan    jane\n");
    printf("First round score %4d %8d %8d\n", cheeta, tarzan, jane);

    return 0;
}
```

许多其他语言都会回避该程序中的三重赋值，但是 C 完全没问题。赋值的顺序是从右往左：首先把 86 赋给 jane，然后再赋给 tarzan，最后赋给 cheeta。因此，程序的输出如下：

```
                  cheetah   tarzan     jane
First round score   68        68        68
```

5.2.2　加法运算符：+

加法运算符（*addition operator*）用于加法运算，使其两侧的值相加。例如，语句：

```
printf("%d", 4 + 20);
```

打印的是 24，而不是表达式

```
4 + 20
```

相加的值（运算对象）可以是变量，也可以是常量。因此，执行下面的语句：

```
income = salary + bribes;
```

计算机会查看加法运算符右侧的两个变量，把它们相加，然后把和赋给变量 income。

在此提醒读者注意，income、salary 和 bribes 都是可修改的左值。因为每个变量都标识了一个可被赋值的数据对象。但是，表达式 salary + brives 是一个右值。

5.2.3　减法运算符：-

减法运算符（*subtraction operator*）用于减法运算，使其左侧的数减去右侧的数。例如，下面的语句把 200.0 赋给 takehome：

```
takehome = 224.00 - 24.00;
```

+和-运算符都被称为*二元运算符*（*binary operator*），即这些运算符需要两个运算对象才能完成操作。

5.2.4　符号运算符：-和+

减号还可用于标明或改变一个值的代数符号。例如，执行下面的语句后，smokey 的值为 12：

```
rocky = -12;
smokey = -rocky;
```

以这种方式使用的负号被称为*一元运算符*（*unary operator*）。一元运算符只需要一个运算对象（见图 5.2）。

C90 标准新增了一元+运算符，它不会改变运算对象的值或符号，只能这样使用：

```
dozen = +12;
```

编译器不会报错。但是在以前，这样做是不允许的。

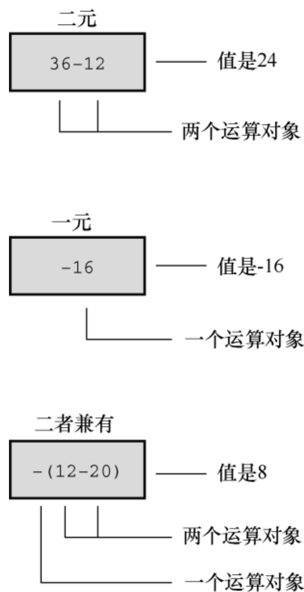


图 5.2　一元和二元运算符

5.2.5　乘法运算符：*

符号*表示乘法。下面的语句用 2.54 乘以 inch，并将结果赋给 cm：

```
cm = 2.54 * inch;
```

C 没有平方函数，如果要打印一个平方表，怎么办？如程序清单 5.4 所示，可以使用乘法来计算平方。

程序清单 5.4　squares.c **程序**

```
/* squares.c -- 计算 1~20 的平方 */
#include <stdio.h>
int main(void)
{
    int num = 1;

    while (num < 21)
    {
        printf("%4d %6d\n", num, num * num);
        num = num + 1;
    }

    return 0;
}
```

该程序打印数字 1～20 及其平方。接下来，我们再看一个更有趣的例子。

1. 指数增长

读者可能听过这样一个故事，一位强大的统治者想奖励做出突出贡献的学者。他问这位学者想要什么，学者指着棋盘说，在第 1 个方格里放 1 粒小麦、第 2 个方格里放 2 粒小麦、第 3 个方格里放 4 粒小麦，第 4 个方格里放 8 粒小麦，以此类推。这位统治者不熟悉数学，很惊讶学者竟然提出如此谦虚的要求。因为他原本准备奖励给学者一大笔财产。如果程序清单 5.5 运行的结果正确，这显然是跟统治者开了一个玩笑。程

序计算出每个方格应放多少小麦，并计算了总数。可能大多数人对小麦的产量不熟悉，该程序以谷粒数为单位，把计算的小麦总数与粗略估计的世界小麦年产量进行了比较。

程序清单 5.5 `wheat.c` **程序**

```
/* wheat.c -- 指数增长 */
#include <stdio.h>
#define SQUARES 64           // 棋盘中的方格数
int main(void)
{
    const double CROP = 2E16;  // 世界小麦年产谷粒数
    double current, total;
    int count = 1;

    printf("square    grains      total     ");
    printf("fraction of \n");
    printf("          added       grains    ");
    printf("world total\n");
    total = current = 1.0;     /* 从 1 颗谷粒开始   */
    printf("%4d %13.2e %12.2e %12.2e\n", count, current,
            total, total / CROP);
    while (count < SQUARES)
    {
        count = count + 1;
        current = 2.0 * current;    /* 下一个方格谷粒翻倍 */
        total = total + current;    /* 更新总数 */
        printf("%4d %13.2e %12.2e %12.2e\n", count, current,
                total, total / CROP);
    }
    printf("That's all.\n");

    return 0;
}
```

程序的输出结果如下：

```
square       grains       total        fraction of
             added        grains       world total
   1         1.00e+00     1.00e+00     5.00e-17
   2         2.00e+00     3.00e+00     1.50e-16
   3         4.00e+00     7.00e+00     3.50e-16
   4         8.00e+00     1.50e+01     7.50e-16
   5         1.60e+01     3.10e+01     1.55e-15
   6         3.20e+01     6.30e+01     3.15e-15
   7         6.40e+01     1.27e+02     6.35e-15
   8         1.28e+02     2.55e+02     1.27e-14
   9         2.56e+02     5.11e+02     2.55e-14
  10         5.12e+02     1.02e+03     5.12e-14
```

10 个方格以后，该学者得到的小麦仅超过了 1000 粒。但是，看看 55 个方格的小麦数是多少：

```
55       1.80e+16     3.60e+16     1.80e+00
```

总量已超过了世界年产量！不妨自己动手运行该程序，看看第 64 个方格有多少小麦。

这个程序示例演示了指数增长的现象。世界人口增长和我们使用的能源都遵循相同的模式。

5.2.6　除法运算符：/

C 使用符号/来表示除法。/左侧的值是被除数，右侧的值是除数。例如，下面 four 的值是 4.0：

```
four = 12.0/3.0;
```

整数除法和浮点数除法不同。浮点数除法的结果是浮点数，而整数除法的结果是整数。整数是没有小数部分的数。这使得 5 除以 3 很让人头痛，因为实际结果有小数部分。在 C 语言中，整数除法结果的小数部分被丢弃，这一过程被称为截断（*truncation*）。

运行程序清单 5.6 中的程序，看看截断的情况，体会整数除法和浮点数除法的区别。

程序清单 5.6　divide.c 程序

```
/* divide.c -- 演示除法 */
#include <stdio.h>
int main(void)
{
    printf("integer division:  5/4   is %d \n", 5 / 4);
    printf("integer division:  6/3   is %d \n", 6 / 3);
    printf("integer division:  7/4   is %d \n", 7 / 4);
    printf("floating division: 7./4. is %1.2f \n", 7. / 4.);
    printf("mixed division:   7./4  is %1.2f \n", 7. / 4);

    return 0;
}
```

程序清单 5.6 中包含一个“混合类型”的示例，即浮点值除以整型值。C 相对其他一些语言而言，在类型管理上比较宽容。尽管如此，一般情况下还是要避免使用混合类型。该程序的输出如下：

```
integer division:  5/4   is 1
integer division:  6/3   is 2
integer division:  7/4   is 1
floating division: 7./4. is 1.75
mixed division:    7./4  is 1.75
```

注意，整数除法会截断计算结果的小数部分（丢弃整个小数部分），不会四舍五入结果。混合整数和浮点数计算的结果是浮点数。实际上，计算机不能真正用浮点数除以整数，编译器会把两个运算对象转换成相同的类型。本例中，在进行除法运算前，整数会被转换成浮点数。

C99 标准以前，C 语言给语言的实现者留有一些空间，让他们来决定如何进行负数的整数除法。一种方法是，舍入过程采用小于或等于浮点数的最大整数。当然，对于 3.8 而言，处理后的 3 符合这一描述。但是-3.8 会怎样？该方法建议四舍五入为-4，因为-4 小于-3.8.但是，另一种舍入方法是直接丢弃小数部分。这种方法被称为“趋零截断”，即把-3.8 转换成-3。在 C99 以前，不同的实现采用不同的方法。但是 C99 规定使用趋零截断。所以，应把-3.8 转换成-3。

5.2.7　运算符优先级

考虑下面的代码：

```
butter = 25.0 + 60.0 * n / SCALE;
```

这条语句中有加法、乘法和除法运算。先算哪一个？是 25.0 加上 60.0，然后把计算的和 85.0 乘以 n，再把结果除以 SCALE？还是 60.0 乘以 n，然后把计算的结果加上 25.0，最后再把结果除以 SCALE？还是其他运算顺序？假设 n 是 6.0，SCALE 是 2.0，带入语句中计算会发现，第 1 种顺序得到的结果是 255，

第 2 种顺序得到的结果是 192.5。C 程序一定是采用了其他的运算顺序，因为程序运行该语句后，butter 的值是 205.0。

显然，执行各种操作的顺序很重要。C 语言对此有明确的规定，通过运算符优先级来解决操作顺序的问题。每个运算符都有自己的优先级。正如普通的算术运算那样，乘法和除法的优先级比加法和减法高，所以先执行乘法和除法。如果两个运算符的优先级相同怎么办？如果它们处理同一个运算对象，则根据它们在语句中出现的顺序来执行。对大多数运算符而言，这种情况都是按从左到右的顺序进行（=运算符除外）。因此，语句：

```
butter = 25.0 + 60.0 * n / SCALE;
```

的运算顺序是：

```
60.0 * n          首先计算表达式中的*或/（假设 n 的值是 6，所以 60.0*n 得 360.0）
360.0 / SCALE     然后计算表达式中第 2 个*或/
25.0 + 180        最后计算表达式里第 1 个+或-，结果为 205.0（假设 SCALE 的值是 2.0）
```

许多人喜欢用表达式树（*expression tree*）来表示求值的顺序，如图 5.3 所示。该图演示了如何从最初的表达式逐步简化为一个值。

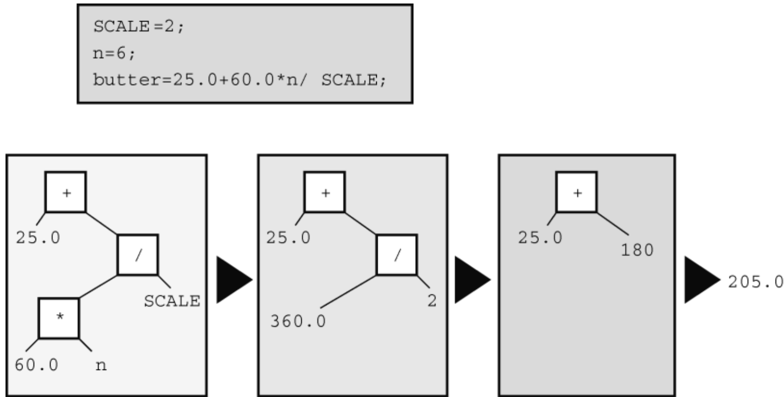


图 5.3　用表达式树演示运算符、运算对象和求值顺序

如何让加法运算在乘法运算之前执行？可以这样做：

```
flour = (25.0 + 60.0 * n) / SCALE;
```

最先执行圆括号中的部分。圆括号内部按正常的规则执行。该例中，先执行乘法运算，再执行加法运算。执行完圆括号内的表达式后，用运算结果除以 SCALE。

表 5.1 总结了到目前为止学过的运算符优先级。

表 5.1　运算符优先级（从低至高）

运算符	结合律
()	从左往右
+ -（一元）	从右往左
* /	从左往右
+ -（二元）	从左往右
=	从右往左

注意正号（加号）和负号（减号）的两种不同用法。结合律栏列出了运算符如何与运算对象结合。例

如，一元负号与它右侧的量相结合，在除法中用除号左侧的运算对象除以右侧的运算对象。

5.2.8　优先级和求值顺序

运算符优先级为表达式中的求值顺序提供重要的依据，但是并没有规定所有的顺序。C 给语言的实现者留出选择的余地。考虑下面的语句：

```
y = 6 * 12 + 5 * 20;
```

当运算符共享一个运算对象时，优先级决定了求值顺序。例如上面的语句中，12 是*和+运算符的运算对象。根据运算符的优先级，乘法的优先级比加法高，所以先进行乘法运算。类似地，先对 5 进行乘法运算而不是加法运算。简而言之，先进行两个乘法运算 6 * 12 和 5 * 20，再进行加法运算。但是，优先级并未规定到底先进行哪一个乘法。C 语言把主动权留给语言的实现者，根据不同的硬件来决定先计算前者还是后者。可能在一种硬件上采用某种方案效率更高，而在另一种硬件上采用另一种方案效率更高。无论采用哪种方案，表达式都会简化为 72 + 100，所以这并不影响最终的结果。但是，读者可能会根据乘法从左往右的结合律，认为应该先执行+运算符左边的乘法。结合律只适用于共享同一运算对象运算符。例如，在表达式 12 / 3 * 2 中，/和*运算符的优先级相同，共享运算对象 3。因此，从左往右的结合律在这种情况起作用。表达式简化为 4 * 2，即 8（如果从右往左计算，会得到 12/6，即 2，这种情况下计算的先后顺序会影响最终的计算结果）。在该例中，两个*运算符并没有共享同一个运算对象，因此从左往右的结合律不适用于这种情况。

学以致用

接下来，我们在更复杂的示例中使用以上规则，请看程序清单 5.7。

程序清单 5.7　rules.c **程序**

```
/* rules.c -- 优先级测试 */
#include <stdio.h>
int main(void)
{
    int top, score;

    top = score = -(2 + 5) * 6 + (4 + 3 * (2 + 3));
    printf("top = %d, score = %d\n", top, score);

    return 0;
}
```

该程序会打印什么值？先根据代码推测一下，再运行程序或阅读下面的分析来检查你的答案。

首先，圆括号的优先级最高。先计算-(2 + 5) * 6 中的圆括号部分，还是先计算(4 + 3 * (2 + 3))中的圆括号部分取决于具体的实现。圆括号的最高优先级意味着，在子表达式-(2 + 5) * 6 中，先计算(2 + 5)的值，得 7。然后，把一元负号应用在 7 上，得-7。现在，表达式是：

```
top = score = -7 * 6 + (4 + 3 * (2 + 3))
```

下一步，计算 2 + 3 的值。表达式变成：

```
top = score = -7 * 6 + (4 + 3 * 5)
```

接下来，因为圆括号中的*比+优先级高，所以表达式变成：

```
top = score = -7 * 6 + (4 + 15)
```

然后，表达式为：

```
top = score = -7 * 6 + 19
```

-7 乘以 6 后，得到下面的表达式：

```
top = score = -42 + 19
```

然后进行加法运算，得到：

```
top = score = -23
```

现在，-23 被赋值给 score，最终 top 的值也是-23。记住，=运算符的结合律是从右往左。

5.3 其他运算符

C 语言有大约 40 个运算符，有些运算符比其他运算符常用得多。前面讨论的是最常用的，本节再介绍 4 个比较有用的运算符。

5.3.1 sizeof 运算符和 size_t 类型

读者在第 3 章就见过 sizeof 运算符。回顾一下，sizeof 运算符以字节为单位返回运算对象的大小（在 C 中，1 字节定义为 char 类型占用的空间大小。过去，1 字节通常是 8 位，但是一些字符集可能使用更大的字节）。运算对象可以是具体的数据对象（如，变量名）或类型。如果运算对象是类型（如，float），则必须用圆括号将其括起来。程序清单 5.8 演示了这两种用法。

程序清单 5.8 sizeof.c 程序

```
// sizeof.c -- 使用sizeof运算符
// 使用C99新增的%zd转换说明 -- 如果编译器不支持%zd，请将其改成%u或%lu
#include <stdio.h>
int main(void)
{
    int n = 0;
    size_t intsize;

    intsize = sizeof (int);
    printf("n = %d, n has %zd bytes; all ints have %zd bytes.\n",
            n, sizeof n, intsize);

    return 0;
}
```

C 语言规定，sizeof 返回 size_t 类型的值。这是一个无符号整数类型，但它不是新类型。前面介绍过，size_t 是语言定义的标准类型。C 有一个 typedef 机制（第 14 章再详细介绍），允许程序员为现有类型创建别名。例如，

```
typedef double real;
```

这样，real 就是 double 的别名。现在，可以声明一个 real 类型的变量：

```
real deal; // 使用typedef
```

编译器查看 real 时会发现，在 typedef 声明中 real 已成为 double 的别名，于是把 deal 创建为 double 类型的变量。类似地，C 头文件系统可以使用 typedef 把 size_t 作为 unsigned int 或 unsigned long 的别名。这样，在使用 size_t 类型时，编译器会根据不同的系统替换标准类型。

C99 做了进一步调整，新增了%zd 转换说明用于 printf()显示 size_t 类型的值。如果系统不支持%zd，可使用%u 或%lu 代替%zd。

5.3.2　求模运算符：%

求模运算符（*modulus operator*）用于整数运算。求模运算符给出其左侧整数除以右侧整数的余数（*remainder*）。例如，`13 % 5`（读作“13 求模 5”）得 3，因为 13 比 5 的两倍多 3，即 13 除以 5 的余数是 3。求模运算符只能用于整数，不能用于浮点数。

乍一看会认为求模运算符像是数学家使用的深奥符号，但是实际上它非常有用。求模运算符常用于控制程序流。例如，假设你正在设计一个账单预算程序，每 3 个月要加进一笔额外的费用。这种情况可以在程序中对月份求模 3（即，`month % 3`），并检查结果是否为 0。如果为 0，便加进额外的费用。等学到第 7 章的 `if` 语句后，读者会更明白。

程序清单 5.9 演示了`%`运算符的另一种用途。同时，该程序也演示了 `while` 循环的另一种用法。

程序清单 5.9　`min_sec.c` 程序

```
// min_sec.c -- 把秒数转换成分和秒
#include <stdio.h>
#define SEC_PER_MIN 60          // 1 分钟 60 秒
int main(void)
{
    int sec, min, left;

    printf("Convert seconds to minutes and seconds!\n");
    printf("Enter the number of seconds (<=0 to quit):\n");
    scanf("%d", &sec);          // 读取秒数
    while (sec > 0)
    {
        min = sec / SEC_PER_MIN;    // 截断分钟数
        left = sec % SEC_PER_MIN;   // 剩下的秒数
        printf("%d seconds is %d minutes, %d seconds.\n", sec,
                min, left);
        printf("Enter next value (<=0 to quit):\n");
        scanf("%d", &sec);
    }
    printf("Done!\n");

    return 0;
}
```

该程序的输出如下：

```
Convert seconds to minutes and seconds!
Enter the number of seconds (<=0 to quit):
154
154 seconds is 2 minutes, 34 seconds.
Enter next value (<=0 to quit):
567
567 seconds is 9 minutes, 27 seconds.
Enter next value (<=0 to quit):
0
Done!
```

程序清单 5.2 使用一个计数器来控制 `while` 循环。当计数器超出给定的大小时，循环终止。而程序清单 5.9 则通过 `scanf()` 为变量 `sec` 获取一个新值。只要该值为正，循环就继续。当用户输入一个 0 或负值时，循环退出。这两种情况设计的要点是，每次循环都会修改被测试的变量值。

负数求模如何进行？C99 规定“趋零截断”之前，该问题的处理方法很多。但自从有了这条规则之后，如果第 1 个运算对象是负数，那么求模的结果为负数；如果第 1 个运算对象是正数，那么求模的结果也是正数：

```
11 / 5 得 2，11 % 5 得 1
11 / -5 得-2，11 % -2 得 1
-11 / -5 得 2，-11 % -5 得-1
-11 / 5 得-2，-11 % 5 得-1
```

如果当前系统不支持 C99 标准，会显示不同的结果。实际上，标准规定：无论何种情况，只要 a 和 b 都是整数值，便可通过 a - (a/b)*b 来计算 a%b。例如，可以这样计算-11%5：

```
-11 - (-11/5) * 5 = -11 -(-2)*5 = -11 -(-10) = -1
```

5.3.3 递增运算符：++

递增运算符（*increment operator*）执行简单的任务，将其运算对象递增 1。该运算符以两种方式出现。第 1 种方式，++出现在其作用的变量前面，这是*前缀*模式；第 2 种方式，++出现在其作用的变量后面，这是*后缀*模式。两种模式的区别在于递增行为发生的时间不同。我们先解释它们的相似之处，再分析它们不同之处。程序清单 5.10 中的程序示例演示了递增运算符是如何工作的。

程序清单 5.10 add_one.c **程序**

```
/* add_one.c -- 递增：前缀和后缀 */
#include <stdio.h>
int main(void)
{
    int ultra = 0, super = 0;

    while (super < 5)
    {
        super++;
        ++ultra;
        printf("super = %d, ultra = %d \n", super, ultra);
    }

    return 0;
}
```

运行该程序后，其输出如下：

```
super = 1, ultra = 1
super = 2, ultra = 2
super = 3, ultra = 3
super = 4, ultra = 4
super = 5, ultra = 5
```

该程序两次同时计数到 5。用下面两条语句分别代替程序中的两条递增语句，程序的输出相同：

```
super = super + 1;
ultra = ultra + 1;
```

这些都是很简单的语句，为何还要创建两个缩写形式？原因之一是，紧凑结构的代码让程序更为简洁，可读性更高。这些运算符让程序看起来很美观。例如，可重写程序清单 5.2（shoes2.c）中的一部分代码：

```
shoe = 3.0;
while (shoe < 18.5)
{
    foot = SCALE * size + ADJUST;
```

```
    printf("%10.1f %20.2f inches\n", shoe, foot);
    ++shoe;
}
```

但是，这样做也没有充分利用递增运算符的优势。还可以这样缩短这段程序：

```
shoe = 2.0;
while (++shoe < 18.5)
{
    foot = SCALE*shoe + ADJUST;
    printf("%10.1f %20.2f inches\n", shoe, foot);
}
```

如上代码所示，把变量的递增过程放入 while 循环的条件中。这种结构在 C 语言中很普遍，我们来仔细分析一下。

首先，这样的 while 循环是如何工作的？很简单。shoe 的值递增 1，然后和 18.5 作比较。如果递增后的值小于 18.5，则执行花括号内的语句一次。然后，shoe 的值再递增 1，重复刚才的步骤，直到 shoe 的值不小于 18.5 为止。注意，我们把 shoe 的初始值从 3.0 改为 2.0，因为在对 foot 第 1 次求值之前，shoe 已经递增了 1（见图 5.4）。

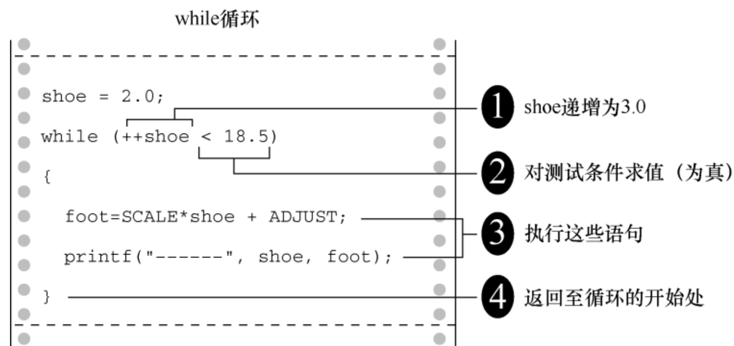


图 5.4　执行一次循环

其次，这样做有什么好处？它使得程序更加简洁。更重要的是，它把控制循环的两个过程集中在一个地方。该循环的主要过程是判断是否继续循环（本例中，要检查鞋子的尺码是否小于 18.5），次要过程是改变待测试的元素（本例中是递增鞋子的尺码）。

如果忘记改变鞋子的尺码，shoe 的值会一直小于 18.5，循环不会停止。计算机将陷入无限循环（*infinite loop*）中，生成无数相同的行。最后，只能强行关闭这个程序。把循环测试和更新循环放在一处，就不会忘记更新循环。

但是，把两个操作合并在一个表达式中，降低了代码的可读性，让代码难以理解。而且，还容易产生计数错误。

递增运算符的另一个优点是，通常它生成的机器语言代码效率更高，因为它和实际的机器语言指令很相似。尽管如此，随着商家推出的 C 编译器越来越智能，这一优势可能会消失。一个智能的编译器可以把 x = x + 1 当作++x 对待。

最后，递增运算符还有一个在某些场合特别有用的特性。我们通过程序清单 5.11 来说明。

程序清单 5.11　post_pre.c 程序

```
/* post_pre.c -- 前缀和后缀 */
#include <stdio.h>
```

```
int main(void)
{
    int a = 1, b = 1;
    int a_post, pre_b;

    a_post = a++;  // 后缀递增
    pre_b = ++b;   // 前缀递增
    printf("a  a_post   b   pre_b \n");
    printf("%1d %5d %5d %5d\n", a, a_post, b, pre_b);

    return 0;
}
```

如果你的编译器没问题，那么程序的输出应该是：

```
a    a_post   b    pre_b
2         1   2        2
```

a 和 b 都递增了 1，但是，a_post 是 a 递增之前的值，而 b_pre 是 b 递增之后的值。这就是++的前缀形式和后缀形式的区别（见图 5.5）。

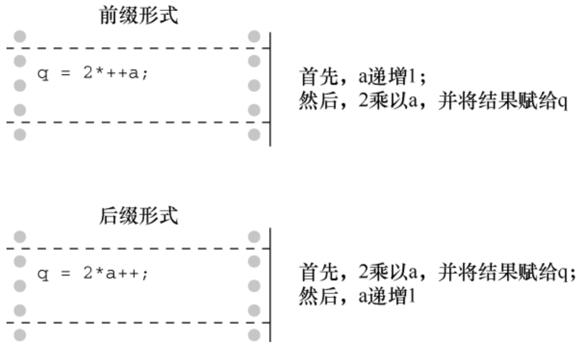


图 5.5　前缀和后缀

```
a_post = a++;      // 后缀：使用 a 的值之后，递增 a
b_pre= ++b;        // 前缀：使用 b 的值之前，递增 b
```

单独使用递增运算符时（如，ego++;），使用哪种形式都没关系。但是，当运算符和运算对象是更复杂表达式的一部分时（如上面的示例），使用前缀或后缀的效果不同。例如，我们曾经建议用下面的代码：

```
while (++shoe < 18.5)
```

该测试条件相当于提供了一个鞋子尺码到 18 的表。如果使用 shoe++而不是++shoes，尺码表会增至 19。因为 shoe 会在与 18.5 进行比较之后才递增，而不是先递增再比较。

当然，使用下面这种形式也没错：

```
shoe = shoe + 1;
```

只不过，有人会怀疑你是否是真正的 C 程序员。

在学习本书的过程中，应多留意使用递增运算符的例子。自己思考是否能互换使用前缀和后缀形式，或者当前环境是否只能使用某种形式。

如果使用前缀形式和后缀形式会对代码产生不同的影响，那么最为明智的是不要那样使用它们。例如，不要使用下面的语句：

```
b = ++i;  // 如果使用 i++，会得到不同的结果
```

应该使用下列语句：

```
++i;      // 第 1 行
b = i;    // 如果第 1 行使用的是 i++，并不会影响 b 的值
```

尽管如此，有时小心翼翼地使用会更有意思。所以，本书会根据实际情况，采用不同的写法。

5.3.4　递减运算符：--

每种形式的递增运算符都有一个递减运算符（*decrement operator*）与之对应，用--代替++即可：

```
--count; // 前缀形式的递减运算符
count--; // 后缀形式的递减运算符
```

程序清单 5.12 演示了计算机可以是位出色的填词家。

程序清单 5.12　`bottles.c` **程序**

```
#include <stdio.h>
#define MAX 100
int main(void)
{
    int count = MAX + 1;

    while (--count > 0) {
        printf("%d bottles of spring water on the wall, "
                 "%d bottles of spring water!\n", count, count);
        printf("Take one down and pass it around,\n");
        printf("%d bottles of spring water!\n\n", count - 1);
    }

    return 0;
}
```

该程序的输出如下（篇幅有限，省略了中间大部分输出）：

```
100 bottles of spring water on the wall, 100 bottles of spring water!
Take one down and pass it around,
99 bottles of spring water!

99 bottles of spring water on the wall, 99 bottles of spring water!
Take one down and pass it around,
98 bottles of spring water!
...
1 bottles of spring water on the wall, 1 bottles of spring water!
Take one down and pass it around,
0 bottles of spring water!
```

显然，这位填词家在复数的表达上有点问题。在学完第 7 章中的条件运算符后，可以解决这个问题。

顺带一提，>运算符表示“大于”，<运算符表示“小于”，它们都是关系运算符（*relational operator*）。我们将在第 6 章中详细介绍关系运算符。

5.3.5　优先级

递增运算符和递减运算符都有很高的结合优先级，只有圆括号的优先级比它们高。因此，`x*y++`表示的是`(x)*(y++)`，而不是`(x+y)++`。不过后者无效，因为递增和递减运算符只能影响一个变量（或者，更普遍地说，只能影响一个可修改的左值），而组合 `x*y` 本身不是可修改的左值。

不要混淆这两个运算符的优先级和它们的求值顺序。假设有如下语句：

```
y = 2;
n = 3;
nextnum = (y + n++)*6;
```

nextnum 的值是多少？把 y 和 n 的值带入上面的第 3 条语句得：

```
nextnum = (2 + 3)*6 = 5*6 = 30
```

n 的值只有在被使用之后才会递增为 4。根据优先级的规定，++只作用于 n，不作用与 y + n。除此之外，根据优先级可以判断何时使用 n 的值对表达式求值，而递增运算符的性质决定了何时递增 n 的值。

如果 n++是表达式的一部分，可将其视为“先使用 n，再递增”；而++n 则表示“先递增 n，再使用”。

5.3.6 不要自作聪明

如果一次用太多递增运算符，自己都会糊涂。例如，利用递增运算符改进 squares.c 程序（程序清单 5.4），用下面的 while 循环替换原程序中的 while 循环：

```
while (num < 21)
     {
     printf("%10d %10d\n", num, num*num++);
     }
```

这个想法看上去不错。打印 num，然后计算 num*num 得到平方值，最后把 num 递增 1。但事实上，修改后的程序只能在某些系统上能正常运行。该程序的问题是：当 printf()获取待打印的值时，可能先对最后一个参数（num*num++）求值，这样在获取其他参数的值之前就递增了 num。所以，本应打印：

```
5          25
```

却打印成：

```
6          25
```

它甚至可能从右往左执行，对最右边的 num（++作用的 num）使用 5，对第 2 个 num 和最左边的 num 使用 6，结果打印出：

```
6          30
```

在 C 语言中，编译器可以自行选择先对函数中的哪个参数求值。这样做提高了编译器的效率，但是如果在函数的参数中使用了递增运算符，就会有一些问题。

类似这样的语句，也会导致一些麻烦：

```
ans = num/2 + 5*(1 + num++);
```

同样，该语句的问题是：编译器可能不会按预想的顺序来执行。你可能认为，先计算第 1 项（num/2），接着计算第 2 项（5*(1 + num++)）。但是，编译器可能先计算第 2 项，递增 num，然后在 num/2 中使用 num 递增后的新值。因此，无法保证编译器到底先计算哪一项。

还有一种情况，也不确定：

```
n = 3;
y = n++ + n++;
```

可以肯定的是，执行完这两条语句后，n 的值会比旧值大 2。但是，y 的值不确定。在对 y 求值时，编译器可以使用 n 的旧值（3）两次，然后把 n 递增 1 两次，这使得 y 的值为 6，n 的值为 5。或者，编译器使用 n 的旧值（3）一次，立即递增 n，再对表达式中的第 2 个 n 使用递增后的新值，然后再递增 n，这使得 y 的值为 7，n 的值为 5。两种方案都可行。对于这种情况更精确地说，结果是未定义的，这意味着 C 标准并未定义结果应该是什么。

遵循以下规则，很容易避免类似的问题：

- 如果一个变量出现在一个函数的多个参数中，不要对该变量使用递增或递减运算符；
- 如果一个变量多次出现在一个表达式中，不要对该变量使用递增或递减运算符。

另一方面，对于何时执行递增，C 还是做了一些保证。我们在本章后面的“副作用和序列点”中学到序列点时再来讨论这部分内容。

5.4 表达式和语句

在前几章中，我们已经多次使用了术语*表达式*（*expression*）和*语句*（*statement*）。现在，我们来进一步学习它们。C 的基本程序步骤由语句组成，而大多数语句都由表达式构成。因此，我们先学习表达式。

5.4.1 表达式

表达式（*expression*）由运算符和运算对象组成（前面介绍过，运算对象是运算符操作的对象）。最简单的表达式是一个单独的运算对象，以此为基础可以建立复杂的表达式。下面是一些表达式：

```
4
-6
4+21
a*(b + c/d)/20
q = 5*2
x = ++q % 3
q > 3
```

如你所见，运算对象可以是常量、变量或二者的组合。一些表达式由*子表达式*（*subexpression*）组成（子表达式即较小的表达式）。例如，`c/d` 是上面例子中 `a*(b + c/d)/20` 的子表达式。

每个表达式都有一个值

C 表达式的一个最重要的特性是，每个表达式都有一个值。要获得这个值，必须根据运算符优先级规定的顺序来执行操作。在上面我们列出的表达式中，前几个都很清晰明了。但是，有赋值运算符（`=`）的表达式的值是什么？这些表达式的值与赋值运算符左侧变量的值相同。因此，表达式 `q = 5*2` 作为一个整体的值是 `10`。那么，表达式 `q > 3` 的值是多少？这种关系表达式的值不是 `0` 就是 `1`，如果条件为真，表达式的值为 `1`；如果条件为假，表达式的值为 `0`。表 5.2 列出了一些表达式及其值：

表 5.2　一些表达式及其值

表达式	值
`-4 + 6`	2
`c = 3 + 8`	11
`5 > 3`	1
`6 + (c = 3 + 8)`	17

虽然最后一个表达式看上去很奇怪，但是在 C 中完全合法（但不建议使用），因为它是两个子表达式的和，每个子表达式都有一个值。

5.4.2 语句

语句（*statement*）是 C 程序的基本构建块。一条语句相当于一条完整的计算机指令。在 C 中，大部分语句都以分号结尾。因此，

```
legs = 4
```

只是一个表达式（它可能是一个较大表达式的一部分），而下面的代码则是一条语句：

```
legs = 4;
```

最简单的语句是空语句：

```
;      //空语句
```

C 把末尾加上一个分号的表达式都看作是一条语句（即，表达式语句）。因此，像下面这样写也没问题：

```
8;
3 + 4;
```

但是，这些语句在程序中什么也不做，不算是真正有用的语句。更确切地说，语句可以改变值或调用函数：

```
x = 25;
++x;
y = sqrt(x);
```

虽然一条语句（或者至少是一条有用的语句）相当于一条完整的指令，但并不是所有的指令都是语句。考虑下面的语句：

```
x = 6 + (y = 5);
```

该语句中的子表达式 `y = 5` 是一条完整的指令，但是它只是语句的一部分。因为一条完整的指令不一定是一条语句，所以分号用于识别在这种情况下的语句（即，简单语句）。

到目前为止，读者已经见过多种语句（不包括空语句）。程序清单 5.13 演示了一些常见的语句。

程序清单 5.13　addemup.c 程序

```
/* addemup.c -- 几种常见的语句 */
#include <stdio.h>
int main(void)                /* 计算前 20 个整数的和    */
{
    int count, sum;           /* 声明[1]                 */

    count = 0;                /* 表达式语句              */
    sum = 0;                  /* 表达式语句              */
    while (count++ < 20)      /* 迭代语句                */
        sum = sum + count;
    printf("sum = %d\n", sum); /* 表达式语句[2]          */

    return 0;             /* 跳转语句                    */
}
```

下面我们讨论程序清单 5.13。到目前为止，相信读者已经很熟悉声明了。尽管如此，我们还是要提醒读者：声明创建了名称和类型，并为其分配内存位置。注意，声明不是表达式语句。也就是说，如果删除声明后面的分号，剩下的部分不是一个表达式，也没有值：

```
int port /* 不是表达式，没有值 */
```

赋值表达式语句在程序中很常用：它为变量分配一个值。赋值表达式语句的结构是，一个变量名，后面是一个赋值运算符，再跟着一个表达式，最后以分号结尾。注意，在 while 循环中有一个赋值表达式语句。赋值表达式语句是表达式语句的一个示例。

[1] 根据 C 标准，声明不是语句。这与 C++有所不同。——译者注

[2] 在 C 语言中，赋值和函数调用都是表达式。没有所谓的“赋值语句”和“函数调用语句”，这些语句实际上都是表达式语句。本书将“assignment statement”均译为“赋值表达式语句”，以提醒读者注意。——译者注

函数表达式语句会引起函数调用。在该例中，调用 printf()函数打印结果。while 语句有 3 个不同的部分（见图 5.6）。首先是关键字 while；然后，圆括号中是待测试的条件；最后如果测试条件为真，则执行 while 循环体中的语句。该例的 while 循环中只有一条语句。可以是本例那样的一条语句，不需要用花括号括起来，也可以像其他例子中那样包含多条语句。多条语句需要用花括号括起来。这种语句是复合语句，稍后马上介绍。

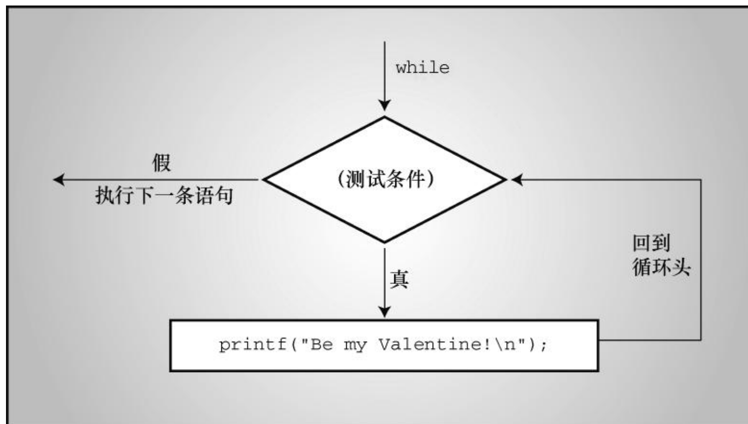


图 5.6　简单的 while 循环结构

while 语句是一种迭代语句，有时也被称为*结构化语句*，因为它的结构比简单的赋值表达式语句复杂。在后面的章节里，我们会遇到许多这样的语句。

副作用和序列点

我们再讨论一个 C 语言的术语*副作用*（*side effect*）。副作用是对数据对象或文件的修改。例如，语句：

```
states = 50;
```

它的副作用是将变量的值设置为 50。副作用？这似乎更像是主要目的！但是从 C 语言的角度看，主要目的是对表达式求值。给出表达式 4 + 6，C 会对其求值得 10；给出表达式 states = 50，C 会对其求值得 50。对该表达式求值的副作用是把变量 states 的值改为 50。跟赋值运算符一样，递增和递减运算符也有副作用，使用它们的主要目的就是使用其副作用。

类似地，调用 printf()函数时，它显示的信息其实是副作用（printf()的返回值是待显示字符的个数）。

序列点（*sequence point*）是程序执行的点，在该点上，所有的副作用都在进入下一步之前发生。在 C 语言中，语句中的分号标记了一个序列点。意思是，在一个语句中，赋值运算符、递增运算符和递减运算符对运算对象做的改变必须在程序执行下一条语句之前完成。后面我们要讨论的一些运算符也有序列点。另外，任何一个完整表达式的结束也是一个序列点。

什么是完整表达式？所谓*完整表达式*（*full expression*），就是指这个表达式不是另一个更大表达式的子表达式。例如，表达式语句中的表达式和 while 循环中的作为测试条件的表达式，都是完整表达式。

序列点有助于分析后缀递增何时发生。例如，考虑下面的代码：

```
while (guests++ < 10)
    printf("%d \n", guests);
```

对于该例，C 语言的初学者认为“先使用值，再递增它”的意思是，在 printf()语句中先使用 guests，再递增它。但是，表达式 guests++ < 10 是一个完整的表达式，因为它是 while 循环的测试条件，所以

该表达式的结束就是一个序列点。因此，C 保证了在程序转至执行 printf()之前发生副作用（即，递增 guests）。同时，使用后缀形式保证了 guests 在完成与 10 的比较后才进行递增。

现在，考虑下面这条语句：

```
y = (4 + x++) + (6 + x++);
```

表达式 4 + x++不是一个完整的表达式，所以 C 无法保证 x 在子表达式 4 + x++求值后立即递增 x。这里，完整表达式是整个赋值表达式语句，分号标记了序列点。所以，C 保证程序在执行下一条语句之前递增 x 两次。C 并未指明是在对子表达式求值以后递增 x，还是对所有表达式求值后再递增 x。因此，要尽量避免编写类似的语句。

5.4.3 复合语句（块）

复合语句（*compound statement*）是用花括号括起来的一条或多条语句，复合语句也称为块（*block*）。shoes2.c 程序使用块让 while 语句包含多条语句。比较下面两个程序段：

```
/* 程序段 1 */
index = 0;
while (index++ < 10)
    sam = 10 * index + 2;
printf("sam = %d\n", sam);

/* 程序段 2 */
index = 0;
while (index++ < 10)
{
    sam = 10 * index + 2;
    printf("sam = %d\n", sam);
}
```

程序段 1，while 循环中只有一条赋值表达式语句。没有花括号，while 语句从 while 这行运行至下一个分号。循环结束后，printf()函数只会被调用一次。

程序段 2，花括号确保两条语句都是 while 循环的一部分，每执行一次循环就调用一次 printf()函数。根据 while 语句的结构，整个复合语句被视为一条语句（见图 5.7）。

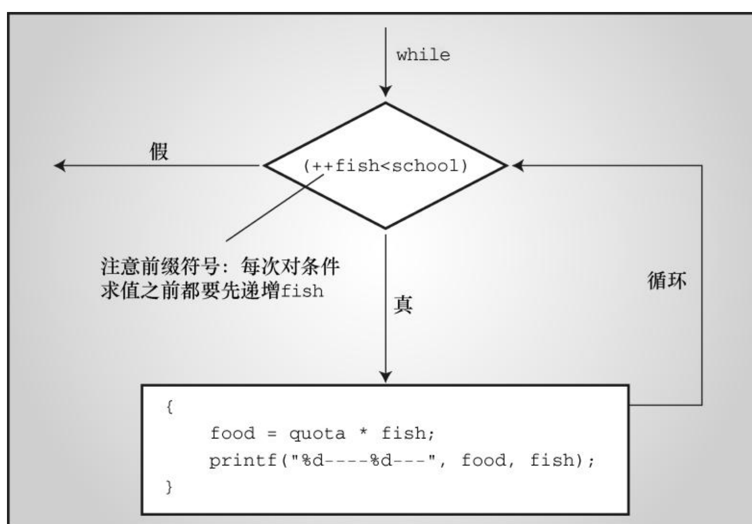


图 5.7 带复合语句的 while 循环

提示　风格提示

再看一下前面的两个 while 程序段，注意循环体中的缩进。缩进对编译器不起作用，编译器通过花括号和 while 循环的结构来识别和解释指令。这里，缩进是为了让读者一眼就可以看出程序是如何组织的。

程序段 2 中，块或复合语句放置花括号的位置是一种常见的风格。另一种常用的风格是:

```
while (index++ < 10) {
     sam = 10*index + 2;
     printf("sam = %d \n", sam);
}
```

这种风格突出了块附属于 while 循环，而前一种风格则强调语句形成一个块。对编译器而言，这两种风格完全相同。

总而言之，使用缩进可以为读者指明程序的结构。

总结　表达式和语句

表达式：

表达式由运算符和运算对象组成。最简单的表达式是不带运算符的一个常量或变量（如，22 或 beebop）。更复杂的例子是 55 + 22 和 vap = 2 * (vip + (vup = 4))。

语句：

到目前为止，读者接触到的语句可分为简单语句和复合语句。简单语句以一个分号结尾。如下所示:

```
赋值表达式语句：      toes = 12;
函数表达式语句：      printf("%d\n", toes);
空语句：              ;    /* 什么也不做 */
```

复合语句（或块）由花括号括起来的一条或多条语句组成。如下面的 while 语句所示:

```
while (years < 100)
{
     wisdom = wisdom * 1.05;
     printf("%d %d\n", years, wisdom);
     years = years + 1;
}
```

5.5　类型转换

通常，在语句和表达式中应使用类型相同的变量和常量。但是，如果使用混合类型，C 不会像 Pascal 那样停在那里死掉，而是采用一套规则进行自动类型转换。虽然这很便利，但是有一定的危险性，尤其是在无意间混合使用类型的情况下（许多 UNIX 系统都使用 lint 程序检查类型“冲突”。如果选择更高错误级别，许多非 UNIX C 编译器也可能报告类型问题）。最好先了解一些基本的类型转换规则。

1．当类型转换出现在表达式时，无论是 unsigned 还是 signed 的 char 和 short 都会被自动转换成 int，如有必要会被转换成 unsigned int（如果 short 与 int 的大小相同，unsigned short 就比 int 大。这种情况下，unsigned short 会被转换成 unsigned int）。在 K&R 那时的 C 中，float 会被自动转换成 double（目前的 C 不是这样）。由于都是从较小类型转换为较大类型，所以这些转换被称为升级（*promotion*）。

2．涉及两种类型的运算，两个值会被分别转换成两种类型的更高级别。

3．类型的级别从高至低依次是 long double、double、float、unsignedlong long、long long、unsigned long、long、unsigned int、int。例外的情况是，当 long 和 int 的大小相同时，unsigned int 比 long 的级别高。之所以 short 和 char 类型没有列出，是因为它们已经被升级到 int 或 unsigned int。

4．在赋值表达式语句中，计算的最终结果会被转换成被赋值变量的类型。这个过程可能导致类型升级或降级（*demotion*）。所谓降级，是指把一种类型转换成更低级别的类型。

5．当作为函数参数传递时，char 和 short 被转换成 int，float 被转换成 double。第 9 章将介绍，函数原型会覆盖自动升级。

类型升级通常都不会有什么问题，但是类型降级会导致真正的麻烦。原因很简单：较低类型可能放不下整个数字。例如，一个 8 位的 char 类型变量储存整数 101 没问题，但是存不下 22334。

如果待转换的值与目标类型不匹配怎么办？这取决于转换涉及的类型。待赋值的值与目标类型不匹配时，规则如下。

1．目标类型是无符号整型，且待赋的值是整数时，额外的位将被忽略。例如，如果目标类型是 8 位 unsigned char，待赋的值是原始值求模 256。

2．如果目标类型是一个有符号整型，且待赋的值是整数，结果因实现而异。

3．如果目标类型是一个整型，且待赋的值是浮点数，该行为是未定义的。

如果把一个浮点值转换成整数类型会怎样？当浮点类型被降级为整数类型时，原来的浮点值会被截断。例如，23.12 和 23.99 都会被截断为 23，-23.5 会被截断为-23。

程序清单 5.14 演示了这些规则。

程序清单 5.14　convert.c 程序

```
/* convert.c -- 自动类型转换 */
#include <stdio.h>
int main(void)
{
    char ch;
    int i;
    float fl;

    fl = i = ch = 'C';                                  /* 第 9 行  */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* 第 10 行 */
    ch = ch + 1;                                        /* 第 11 行 */
    i = fl + 2 * ch;                                    /* 第 12 行 */
    fl = 2.0 * ch + i;                                  /* 第 13 行 */
    printf("ch = %c, i = %d, fl = %2.2f\n", ch, i, fl); /* 第 14 行 */
    ch = 1107;                                          /* 第 15 行 */
    printf("Now ch = %c\n", ch);                        /* 第 16 行 */
    ch = 80.89;                                         /* 第 17 行 */
    printf("Now ch = %c\n", ch);                        /* 第 18 行 */

    return 0;
}
```

运行 convert.c 后输出如下：

```
ch = C, i = 67, fl = 67.00
ch = D, i = 203, fl = 339.00
Now ch = S
Now ch = P
```

在我们的系统中，char 是 8 位，int 是 32 位。程序的分析如下。

- 第 9 行和第 10 行：字符'C'被作为 1 字节的 ASCII 值储存在 ch 中。整数变量 i 接受由'C'转换的整数，即按 4 字节储存 67。最后，fl 接受由 67 转换的浮点数 67.00。
- 第 11 行和第 14 行：字符变量'C'被转换成整数 67，然后加 1。计算结果是 4 字节整数 68，被截断成 1 字节储存在 ch 中。根据%c 转换说明打印时，68 被解释成'D'的 ASCII 码。
- 第 12 行和第 14 行：ch 的值被转换成 4 字节的整数（68），然后 2 乘以 ch。为了和 fl 相加，乘积整数（136）被转换成浮点数。计算结果（203.00f）被转换成 int 类型，并储存在 i 中。
- 第 13 行和第 14 行：ch 的值（'D'，或 68）被转换成浮点数，然后 2 乘以 ch。为了做加法，i 的值（203）被转换为浮点类型。计算结果（339.00）被储存在 fl 中。
- 第 15 行和第 16 行：演示了类型降级的示例。把 ch 设置为一个超出其类型范围的值，忽略额外的位后，最终 ch 的值是字符 S 的 ASCII 码。或者，更确切地说，ch 的值是 1107 % 265，即 83。
- 第 17 行和第 18 行：演示了另一个类型降级的示例。把 ch 设置为一个浮点数，发生截断后，ch 的值是字符 P 的 ASCII 码。

5.5.1　强制类型转换运算符

通常，应该避免自动类型转换，尤其是类型降级。但是如果能小心使用，类型转换也很方便。我们前面讨论的类型转换都是自动完成的。然而，有时需要进行精确的类型转换，或者在程序中表明类型转换的意图。这种情况下要用到强制类型转换（*cast*），即在某个量的前面放置用圆括号括起来的类型名，该类型名即是希望转换成的目标类型。圆括号和它括起来的类型名构成了强制类型转换运算符（*cast operator*），其通用形式是：

(*type*)

用实际需要的类型（如，long）替换 *type* 即可。

考虑下面两行代码，其中 mice 是 int 类型的变量。第 2 行包含两次 int 强制类型转换。

```
mice = 1.6 + 1.7;
mice = (int)1.6 + (int)1.7;
```

第 1 行使用自动类型转换。首先，1.6 和 1.7 相加得 3.3。然后，为了匹配 int 类型的变量，3.3 被类型转换截断为整数 3。第 2 行，1.6 和 1.7 在相加之前都被转换成整数（1），所以把 1+1 的和赋给变量 mice。本质上，两种类型转换都好不到哪里去，要考虑程序的具体情况再做取舍。

一般而言，不应该混合使用类型（因此有些语言直接不允许这样做），但是偶尔这样做也是有用的。C 语言的原则是避免给程序员设置障碍，但是程序员必须承担使用的风险和责任。

总结　C 的一些运算符

下面是我们学过的一些运算符。

赋值运算符：

=　　将其右侧的值赋给左侧的变量

算术运算符：

+	将其左侧的值与右侧的值相加
-	将其左侧的值减去右侧的值
-	作为一元运算符，改变其右侧值的符号
*	将其左侧的值乘以右侧的值
/	将其左侧的值除以右侧的值，如果两数都是整数，计算结果将被截断
%	当其左侧的值除以右侧的值时，取其余数（只能应用于整数）
++	对其右侧的值加 1（前缀模式），或对其左侧的值加 1（后缀模式）
--	对其右侧的值减 1（前缀模式），或对其左侧的值减 1（后缀模式）

其他运算符：

sizeof	获得其右侧运算对象的大小（以字节为单位），运算对象可以是一个被圆括号括起来的类型说明符，如 sizeof(float)，或者是一个具体的变量名、数组名等，如 sizeof foo
(类型名)	强制类型转换运算符将其右侧的值转换成圆括号中指定的类型，如(float)9 把整数 9 转换成浮点数 9.0

5.6　带参数的函数

现在，相信读者已经熟悉了带参数的函数。要掌握函数，还要学习如何编写自己的函数（在此之前，读者可能要复习一下程序清单 2.3 中的 butler() 函数，该函数不带任何参数）。程序清单 5.15 中有一个 pound() 函数，打印指定数量的#号（该符号也叫作编号符号或井号）。该程序还演示了类型转换的应用。

程序清单 5.15　pound.c **程序**

```
/* pound.c -- 定义一个带一个参数的函数   */
#include <stdio.h>
void pound(int n);// ANSI 函数原型声明
int main(void)
{
    int times = 5;
    char ch = '!';     // ASCII 码是 33
    float f = 6.0f;

    pound(times);      // int 类型的参数
    pound(ch);         // 和 pound((int)ch);相同
    pound(f);          // 和 pound((int)f);相同

    return 0;
}

void pound(int n)      // ANSI 风格函数头
{                      // 表明该函数接受一个 int 类型的参数
    while (n-- > 0)
        printf("#");
    printf("\n");
}
```

运行该程序后，输出如下：

```
#####
###################################
######
```

首先，看程序的函数头：

```
void pound(int n)
```

如果函数不接受任何参数，函数头的圆括号中应该写上关键字 void。由于该函数接受一个 int 类型的参数，所以圆括号中包含一个 int 类型变量 n 的声明。参数名应遵循 C 语言的命名规则。

声明参数就创建了被称为形式参数（*formal argument* 或 *formal parameter*，简称形参）的变量。该例中，形式参数是 int 类型的变量 n。像 pound(10)这样的函数调用会把 10 赋给 n。在该程序中，调用 pound(times)就是把 times 的值（5）赋给 n。我们称函数调用传递的值为实际参数（*actual argument* 或 *actual parameter*），简称实参。所以，函数调用 pound(10)把实际参数 10 传递给函数，然后该函数把 10 赋给形式参数（变量 n）。也就是说，main()中的变量 times 的值被拷贝给 pound()中的新变量 n。

注意　实参和形参

在英文中，argument 和 parameter 经常可以互换使用，但是 C99 标准规定了：对于 actual argument 或 actual parameter 使用术语 argument（译为实参）；对于 formal argument 或 formal parameter 使用术语 parameter（译为形参）。为遵循这一规定，我们可以说形参是变量，实参是函数调用提供的值，实参被赋给相应的形参。因此，在程序清单 5.15 中，times 是 pound()的实参，n 是 pound()的形参。类似地，在函数调用 pound(times + 4)中，表达式 times + 4 的值是该函数的实参。

变量名是函数私有的，即在函数中定义的函数名不会和别处的相同名称发生冲突。如果在 pound()中用 times 代替 n，那么这个 times 与 main()中的 times 不同。也就是说，程序中出现了两个同名的变量，但是程序可以区分它们。

现在，我们来学习函数调用。第 1 个函数调用是 pound(times)，times 的值 5 被赋给 n。因此，printf()函数打印了 5 个井号和 1 个换行符。第 2 个函数调用是 pound(ch)。这里，ch 是 char 类型，被初始化为!字符，在 ASCII 中 ch 的数值是 33。但是 pound()函数的参数类型是 int，与 char 不匹配。程序开头的函数原型在这里发挥了作用。原型（*prototype*）即是函数的声明，描述了函数的返回值和参数。pound()函数的原型说明了两点：

- 该函数没有返回值（函数名前面有 void 关键字）；
- 该函数有一个 int 类型的参数。

该例中，函数原型告诉编译器 pound()需要一个 int 类型的参数。相应地，当编译器执行到 pound(ch)表达式时，会把参数 ch 自动转换成 int 类型。在我们的系统中，该参数从 1 字节的 33 变成 4 字节的 33，所以现在 33 的类型满足函数的要求。与此类似，最后一次调用是 pound(f)，使得 float 类型的变量被转换成合适的类型。

在 ANSI C 之前，C 使用的是函数声明，而不是函数原型。函数声明只指明了函数名和返回类型，没有指明参数类型。为了向下兼容，C 现在仍然允许这样的形式：

```
void pound(); /* ANSI C之前的函数声明 */
```

如果用这条函数声明代替 pound.c 程序中的函数原型会怎样？第 1 次函数调用，pound(times)没问题，因为 times 是 int 类型。第 2 次函数调用，pound(ch)也没问题，因为即使缺少函数原型，C 也会把 char 和 short 类型自动升级为 int 类型。第 3 次函数调用，pound(f)会失败，因为缺少函数原型，

float 会被自动升级为 double，这没什么用。虽然程序仍然能运行，但是输出的内容不正确。在函数调用中显式使用强制类型转换，可以修复这个问题：

```
pound ((int)f); // 把 f 强制类型转换为正确的类型
```

注意，如果 f 的值太大，超过了 int 类型表示的范围，这样做也不行。

5.7 示例程序

程序清单 5.16 演示了本章介绍的几个概念，这个程序对某些人很有用。程序看起来很长，但是所有的计算都在程序的后面几行中。我们尽量使用大量的注释，让程序看上去清晰明了。请通读该程序，稍后我们会分析几处要点。

程序清单 5.16 running.c **程序**

```
// running.c -- A useful program for runners
#include <stdio.h>
const int S_PER_M = 60;            // 1 分钟的秒数
const int S_PER_H = 3600;          // 1 小时的分钟数
const double M_PER_K = 0.62137;    // 1 公里的英里数
int main(void)
{
    double distk, distm;   // 跑过的距离（分别以公里和英里为单位）
    double rate;           // 平均速度（以英里/小时为单位）
    int min, sec;          // 跑步用时（以分钟和秒为单位）
    int time;              // 跑步用时（以秒为单位）
    double mtime;          // 跑 1 英里需要的时间，以秒为单位
    int mmin, msec;        // 跑 1 英里需要的时间，以分钟和秒为单位

    printf("This program converts your time for a metric race\n");
    printf("to a time for running a mile and to your average\n");
    printf("speed in miles per hour.\n");
    printf("Please enter, in kilometers, the distance run.\n");
    scanf("%lf", &distk);          // %lf 表示读取一个 double 类型的值
    printf("Next enter the time in minutes and seconds.\n");
    printf("Begin by entering the minutes.\n");
    scanf("%d", &min);
    printf("Now enter the seconds.\n");
    scanf("%d", &sec);

    time = S_PER_M * min + sec;     // 把时间转换成秒
    distm = M_PER_K * distk;        // 把公里转换成英里
    rate = distm / time * S_PER_H;  // 英里/秒×秒/小时 = 英里/小时
    mtime = (double) time / distm;  // 时间/距离 = 跑 1 英里所用的时间
    mmin = (int) mtime / S_PER_M;   // 求出分钟数
    msec = (int) mtime % S_PER_M;   // 求出剩余的秒数

    printf("You ran %1.2f km (%1.2f miles) in %d min, %d sec.\n",
           distk, distm, min, sec);
    printf("That pace corresponds to running a mile in %d min, ",
           mmin);
    printf("%d sec.\nYour average speed was %1.2f mph.\n", msec,
```

```
               rate);

    return 0;
}
```

程序清单 5.16 使用了 min_sec 程序（程序清单 5.9）中的方法把时间转换成分钟和秒，除此之外还使用了类型转换。为什么要进行类型转换？因为程序在秒转换成分钟的部分需要整型参数，但是在公里转换成英里的部分需要浮点运算。我们使用强制类型转换运算符进行了显式转换。

实际上，我们曾经利用自动类型转换编写这个程序，即使用 int 类型的 mtime 来强制时间计算转换成整数形式。但是，在测试的 11 个系统中，这个版本的程序在 1 个系统上无法运行，这是由于编译器（版本比较老）没有遵循 C 规则。而使用强制类型转换就没有问题。对读者而言，强制类型转换强调了转换类型的意图，对编译器而言也是如此。

下面是程序清单 5.16 的输出示例：

```
This program converts your time for a metric race
to a time for running a mile and to your average
speed in miles per hour.
Please enter, in kilometers, the distance run.
10.0
Next enter the time in minutes and seconds.
Begin by entering the minutes.
36
Now enter the seconds.
23
You ran 10.00 km (6.21 miles) in 36 min, 23 sec.
That pace corresponds to running a mile in 5 min, 51 sec.
Your average speed was 10.25 mph.
```

5.8 关键概念

C 通过运算符提供多种操作。每个运算符的特性包括运算对象的数量、优先级和结合律。当两个运算符共享一个运算对象时，优先级和结合律决定了先进行哪项运算。每个 C 表达式都有一个值。如果不了解运算符的优先级和结合律，写出的表达式可能不合法或者表达式的值与预期不符。这会影响你成为一名优秀的程序员。

虽然 C 允许编写混合数值类型的表达式，但是算术运算要求运算对象都是相同的类型。因此，C 会进行自动类型转换。尽管如此，不要养成依赖自动类型转换的习惯，应该显式选择合适的类型或使用强制类型转换。这样，就不用担心出现不必要的自动类型转换。

5.9 本章小结

C 语言有许多运算符，如本章讨论的赋值运算符和算术运算符。一般而言，运算符需要一个或多个运算对象才能完成运算生成一个值。只需要一个运算对象的运算符（如负号和 sizeof）称为*一元运算符*，需要两个运算对象的运算符（如加法运算符和乘法运算符）称为*二元运算符*。

*表达式*由运算符和运算对象组成。在 C 语言中，每个表达式都有一个值，包括赋值表达式和比较表达式。运算符优先级规则决定了表达式中各项的求值顺序。当两个运算符共享一个运算对象时，先进行优先级高的运算。如果运算符的优先级相等，由结合律（从左往右或从右往左）决定求值顺序。

大部分语句都以分号结尾。最常用的语句是*表达式语句*。用花括号括起来的一条或多条语句构成了*复合语句*（或称为块）。while 语句是一种*迭代语句*，只要测试条件为真，就重复执行循环体中的语句。

在 C 语言中，许多*类型转换*都是自动进行的。当 char 和 short 类型出现在表达式里或作为函数的参数（函数原型除外）时，都会被升级为 int 类型；float 类型在函数参数中时，会被升级为 double 类型。在 K&R C（不是 ANSI C）下，表达式中的 float 也会被升级为 double 类型。当把一种类型的值赋给另一种类型的变量时，值将被转换成与变量的类型相同。当把较大类型转换成较小类型时（如，long 转换成 short，或 double 转换成 float），可能会丢失数据。根据本章介绍的规则，在混合类型的运算中，较小类型会被转换成较大类型。

定义带一个参数的函数时，便在函数定义中声明了一个*变量*，或称为*形式参数*。然后，在函数调用中传入的值会被赋给这个变量。这样，在函数中就可以使用该值了。

5.10 复习题

复习题的参考答案在附录 A 中。

1．假设所有变量的类型都是 int，下列各项变量的值是多少：

a. `x = (2 + 3) * 6;`

b. `x = (12 + 6)/2*3;`

c. `y = x = (2 + 3)/4;`

d. `y = 3 + 2*(x = 7/2);`

2．假设所有变量的类型都是 int，下列各项变量的值是多少：

a. `x = (int)3.8 + 3.3;`

b. `x = (2 + 3) * 10.5;`

c. `x = 3 / 5 * 22.0;`

d. `x = 22.0 * 3 / 5;`

3．对下列各表达式求值：

a. `30.0 / 4.0 * 5.0;`

b. `30.0 / (4.0 * 5.0);`

c. `30 / 4 * 5;`

d. `30 * 5 / 4;`

e. `30 / 4.0 * 5;`

f. `30 / 4 * 5.0;`

4．请找出下面的程序中的错误。

```
int main(void)
{
    int i = 1,
    float n;
    printf("Watch out! Here come a bunch of fractions!\n");
    while (i < 30)
         n = 1/i;
         printf(" %f", n);
    printf("That's all, folks!\n");
    return;
}
```

5．这是程序清单 5.9 的另一个版本。从表面上看，该程序只使用了一条 scanf()语句，比程序清单

5.9 简单。请找出不如原版之处。

```
#include <stdio.h>
#define S_TO_M 60
int main(void)
{
    int sec, min, left;

    printf("This program converts seconds to minutes and ");
    printf("seconds.\n");
    printf("Just enter the number of seconds.\n");
    printf("Enter 0 to end the program.\n");
    while (sec > 0) {
        scanf("%d", &sec);
        min = sec/S_TO_M;
        left = sec % S_TO_M;
        printf("%d sec is %d min, %d sec. \n", sec, min, left);
        printf("Next input?\n");
    }
    printf("Bye!\n");
    return 0;
}
```

6. 下面的程序将打印出什么内容？

```
#include <stdio.h>
#define FORMAT "%s! C is cool!\n"
int main(void)
{
    int num = 10;

    printf(FORMAT,FORMAT);
    printf("%d\n", ++num);
    printf("%d\n", num++);
    printf("%d\n", num--);
    printf("%d\n", num);
    return 0;
}
```

7. 下面的程序将打印出什么内容？

```
#include <stdio.h>
int main(void)
{
    char c1, c2;
    int diff;
    float num;

    c1 = 'S';
    c2 = 'O';
    diff = c1 - c2;
    num = diff;
    printf("%c%c%c:%d %3.2f\n", c1, c2, c1, diff, num);
    return 0;
}
```

8. 下面的程序将打印出什么内容？

```
#include <stdio.h>
#define TEN 10
int main(void)
```

```
{
    int n = 0;

    while (n++ < TEN)
        printf("%5d", n);
    printf("\n");
    return 0;
}
```

9．修改上一个程序，使其可以打印字母 a～g。

10．假设下面是完整程序中的一部分，它们分别打印什么？

a.

```
int x = 0;

while (++x < 3)
    printf("%4d", x);
```

b.

```
int x = 100;

while (x++ < 103)
    printf("%4d\n",x);
    printf("%4d\n",x);
```

c.

```
char ch = 's';

while (ch < 'w')
{
    printf("%c", ch);
    ch++;
}
printf("%c\n",ch);
```

11．下面的程序会打印出什么？

```
#define MESG "COMPUTER BYTES DOG"
#include <stdio.h>
int main(void)
{
    int n = 0;

    while ( n < 5 )
        printf("%s\n", MESG);
        n++;
    printf("That's all.\n");
    return 0;
}
```

12．分别编写一条语句，完成下列各任务（或者说，使其具有以下副作用）：

a．将变量 x 的值增加 10

b．将变量 x 的值增加 1

c．将 a 与 b 之和的两倍赋给 c

d．将 a 与 b 的两倍之和赋给 c

13．分别编写一条语句，完成下列各任务：

a．将变量 x 的值减少 1

b．将 n 除以 k 的余数赋给 m

c．q 除以 b 减去 a，并将结果赋给 p

d．a 与 b 之和除以 c 与 d 的乘积，并将结果赋给 x

5.11　编程练习

1. 编写一个程序，把用分钟表示的时间转换成用小时和分钟表示的时间。使用#define 或 const 创建一个表示 60 的符号常量或 const 变量。通过 while 循环让用户重复输入值，直到用户输入小于或等于 0 的值才停止循环。

2. 编写一个程序，提示用户输入一个整数，然后打印从该数到比该数大 10 的所有整数（例如，用户输入 5，则打印 5～15 的所有整数，包括 5 和 15）。要求打印的各值之间用一个空格、制表符或换行符分开。

3. 编写一个程序，提示用户输入天数，然后将其转换成周数和天数。例如，用户输入 18，则转换成 2 周 4 天。以下面的格式显示结果：

```
18 days are 2 weeks, 4 days.
```

通过 while 循环让用户重复输入天数，当用户输入一个非正值时（如 0 或-20），循环结束。

4. 编写一个程序，提示用户输入一个身高（单位：厘米），并分别以厘米和英寸为单位显示该值，允许有小数部分。程序应该能让用户重复输入身高，直到用户输入一个非正值。其输出示例如下：

```
Enter a height in centimeters: 182
182.0 cm = 5 feet, 11.7 inches
Enter a height in centimeters (<=0 to quit): 168.7
168.0 cm = 5 feet, 6.4 inches
Enter a height in centimeters (<=0 to quit): 0
bye
```

5. 修改程序 addemup.c（程序清单 5.13），你可以认为 addemup.c 是计算 20 天里赚多少钱的程序（假设第 1 天赚$1、第 2 天赚$2、第 3 天赚$3，以此类推）。修改程序，使其可以与用户交互，根据用户输入的数进行计算（即，用读入的一个变量来代替 20）。

6. 修改编程练习 5 的程序，使其能计算整数的平方和（可以认为第 1 天赚$1、第 2 天赚$4、第 3 天赚$9，以此类推，这看起来很不错）。C 没有平方函数，但是可以用 n * n 来表示 n 的平方。

7. 编写一个程序，提示用户输入一个 double 类型的数，并打印该数的立方值。自己设计一个函数计算并打印立方值。main()函数要把用户输入的值传递给该函数。

8. 编写一个程序，显示求模运算的结果。把用户输入的第 1 个整数作为求模运算符的第 2 个运算对象，该数在运算过程中保持不变。用户后面输入的数是第 1 个运算对象。当用户输入一个非正值时，程序结束。其输出示例如下：

```
This program computes moduli.
Enter an integer to serve as the second operand: 256
Now enter the first operand: 438
438 % 256 is 182
Enter next number for first operand (<= 0 to quit): 1234567
1234567 % 256 is 135
Enter next number for first operand (<= 0 to quit): 0
Done
```

9. 编写一个程序，要求用户输入一个华氏温度。程序应读取 double 类型的值作为温度值，并把该值作为参数传递给一个用户自定义的函数 Temperatures()。该函数计算摄氏温度和开氏温度，并

以小数点后面两位数字的精度显示 3 种温度。要使用不同的温标来表示这 3 个温度值。下面是华氏温度转摄氏温度的公式：

摄氏温度 = 5.0 / 9.0 * (华氏温度 - 32.0)

开氏温标常用于科学研究，0 表示绝对零，代表最低的温度。下面是摄氏温度转开氏温度的公式：

开氏温度 = 摄氏温度 + 273.16

`Temperatures()`函数中用 `const` 创建温度转换中使用的变量。在 `main()`函数中使用一个循环让用户重复输入温度，当用户输入 `q` 或其他非数字时，循环结束。`scanf()`函数返回读取数据的数量，所以如果读取数字则返回 `1`，如果读取 `q` 则不返回 `1`。可以使用==运算符将 `scanf()`的返回值和 `1` 作比较，测试两值是否相等。

第 6 章

C 控制语句：循环

本章介绍以下内容：

- 关键字：for、while、do while
- 运算符：<、>、>=、<=、!=、==、+=、*=、-=、/=、%=
- 函数：fabs()
- C 语言有 3 种循环：for、while、do while
- 使用关系运算符构建控制循环的表达式
- 其他运算符
- 循环常用的数组
- 编写有返回值的函数

大多数人都希望自己是体格强健、天资聪颖、多才多艺的能人。虽然有时事与愿违，但至少我们用 C 能写出这样的程序。诀窍是控制程序流。对于计算机科学（是研究计算机，不是用计算机做研究）而言，一门语言应该提供以下 3 种形式的程序流：

- 执行语句序列；
- 如果满足某些条件就重复执行语句序列（循环）；
- 通过测试选择执行哪一个语句序列（分支）。

读者对第一种形式应该很熟悉，前面学过的程序中大部分都是由语句序列组成。while 循环属于第二种形式。本章将详细讲解 while 循环和其他两种循环：for 和 do while。第三种形式用于在不同的执行方案之间进行选择，让程序更“智能”，且极大地提高了计算机的用途。不过，要等到下一章才介绍这部分的内容。本章还将介绍数组，可以把新学的知识应用在数组上。另外，本章还将继续介绍函数的相关内容。首先，我们从 while 循环开始学习。

6.1 再探 while 循环

经过上一章的学习，读者已经熟悉了 while 循环。这里，我们用一个程序来回顾一下，程序清单 6.1 根据用户从键盘输入的整数进行求和。程序利用了 scanf()的返回值来结束循环。

程序清单 6.1 summing.c 程序

```
/* summing.c -- 根据用户键入的整数求和 */
#include <stdio.h>
int main(void)
{
     long num;
     long sum = 0L;          /* 把 sum 初始化为 0   */
```

```
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status == 1)    /* == 的意思是“等于”    */
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

该程序使用 long 类型以储存更大的整数。尽管 C 编译器会把 0 自动转换为合适的类型，但是为了保持程序的一致性，我们把 sum 初始化为 0L（long 类型的 0），而不是 0（int 类型的 0）。

该程序的运行示例如下：

```
Please enter an integer to be summed (q to quit): 44
Please enter next integer (q to quit): 33
Please enter next integer (q to quit): 88
Please enter next integer (q to quit): 121
Please enter next integer (q to quit): q
Those integers sum to 286.
```

6.1.1　程序注释

先看 while 循环，该循环的测试条件是如下表达式：

```
status == 1
```

==运算符是 C 的相等运算符（*equality operator*），该表达式判断 status 是否等于 1。不要把 status == 1 与 status = 1 混淆，后者是把 1 赋给 status。根据测试条件 status == 1，只要 status 等于 1，循环就会重复。每次循环，num 的当前值都被加到 sum 上，这样 sum 的值始终是当前整数之和。当 status 的值不为 1 时，循环结束。然后程序打印 sum 的最终值。

要让程序正常运行，每次循环都要获取 num 的一个新值，并重置 status。程序利用 scanf()的两个不同的特性来完成。首先，使用 scanf()读取 num 的一个新值；然后，检查 scanf()的返回值判断是否成功获取值。第 4 章中介绍过，scanf()返回成功读取项的数量。如果 scanf()成功读取一个整数，就把该数存入 num 并返回 1，随后返回值将被赋给 status（注意，用户输入的值储存在 num 中，不是 status 中）。这样做同时更新了 num 和 status 的值，while 循环进入下一次迭代。如果用户输入的不是数字（如，q），scanf()会读取失败并返回 0。此时，status 的值就是 0，循环结束。因为输入的字符 q 不是数字，所以它会被放回输入队列中（实际上，不仅仅是 q，任何非数值的数据都会导致循环终止，但是提示用户输入 q 退出程序比提示用户输入一个非数字字符要简单）。

如果 scanf()在转换值之前出了问题（例如，检测到文件结尾或遇到硬件问题），会返回一个特殊值 EOF（其值通常被定义为-1）。这个值也会引起循环终止。

如何告诉循环何时停止？该程序利用 scanf()的双重特性避免了在循环中交互输入时的这个棘手的问题。例如，假设 scanf()没有返回值，那么每次循环只会改变 num 的值。虽然可以使用 num 的值来结束循环，比如把 num > 0（num 大于 0）或 num ！= 0（num 不等于 0）作为测试条件，但是这样用户就

不能输入某些值，如-3 或 0。也可以在循环中添加代码，例如每次循环时询问用户“是否继续循环？<y/n>”，然后判断用户是否输入 y。这个方法有些笨拙，而且还减慢了输入的速度。使用 scanf()的返回值，轻松地避免了这些问题。

现在，我们来看看该程序的结构。总结如下：

把 sum 初始化为 0

提示用户输入数据

读取用户输入的数据

当输入的数据为整数时，

　　输入添加给 sum，

　　提示用户进行输入，

　　然后读取下一个输入

输入完成后，打印 sum 的值

顺带一提，这叫作伪代码（*pseudocode*），是一种用简单的句子表示程序思路的方法，它与计算机语言的形式相对应。伪代码有助于设计程序的逻辑。确定程序的逻辑无误之后，再把伪代码翻译成实际的编程代码。使用伪代码的好处之一是，可以把注意力集中在程序的组织和逻辑上，不用在设计程序时还要分心如何用编程语言来表达自己的想法。例如，可以用缩进来代表一块代码，不用考虑 C 的语法要用花括号把这部分代码括起来。

总之，因为 while 循环是入口条件循环，程序在进入循环体之前必须获取输入的数据并检查 status 的值，所以在 while 前面要有一个 scanf()。要让循环继续执行，在循环内需要一个读取数据的语句，这样程序才能获取下一个 status 的值，所以在 while 循环末尾还要有一个 scanf()，它为下一次迭代做好了准备。可以把下面的伪代码作为 while 循环的标准格式：

获得第 1 个用于测试的值

当测试为真时

　　处理值

　　获取下一个值

6.1.2　C 风格读取循环

根据伪代码的设计思路，程序清单 6.1 可以用 Pascal、BASIC 或 FORTRAN 来编写。但是 C 更为简洁，下面的代码：

```
status = scanf("%ld", &num);
while (status == 1)
{
    /* 循环行为 */
    status = scanf("%ld", &num);
}
```

可以用这些代码替换：

```
while (scanf("%ld", &num) == 1)
{
    /*循环行为*/
}
```

第二种形式同时使用 scanf()的两种不同的特性。首先，如果函数调用成功，scanf()会把一个值存入 num。然后，利用 scanf()的返回值（0 或 1，不是 num 的值）控制 while 循环。因为每次迭代都会判断循环的条件，所以每次迭代都要调用 scanf()读取新的 num 值来做判断。换句话说，C 的语法特性让你可以用下面的精简版本替换标准版本：

当获取值和判断值都成功

　　处理该值

接下来，我们正式地学习 while 语句。

6.2　while 语句

while 循环的通用形式如下：

```
while ( expression )
      statement
```

statement 部分可以是以分号结尾的简单语句，也可以是用花括号括起来的复合语句。

到目前为止，程序示例中的 *expression* 部分都使用关系表达式。也就是说，*expression* 是值之间的比较，可以使用任何表达式。如果 *expression* 为真（或者更一般地说，非零），执行 *statement* 部分一次，然后再次判断 *expression*。在 *expression* 为假（0）之前，循环的判断和执行一直重复进行。每次循环都被称为一次迭代（*iteration*），如图 6.1 所示。

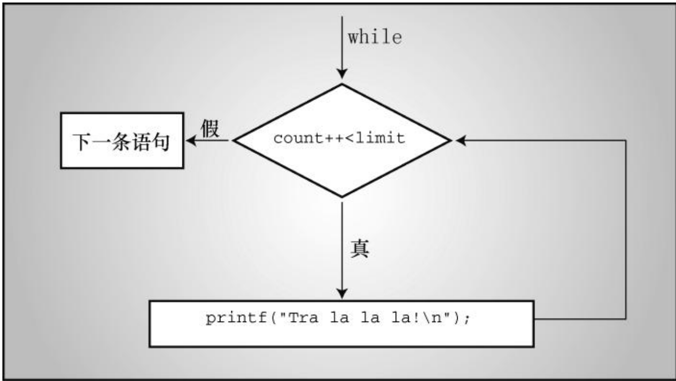


图 6.1　while 循环的结构

6.2.1　终止 while 循环

while 循环有一点非常重要：在构建 while 循环时，必须让测试表达式的值有变化，表达式最终要为假。否则，循环就不会终止（实际上，可以使用 break 和 if 语句来终止循环，但是你尚未学到）。考虑下面的例子：

```
index = 1;
while (index < 5)
    printf("Good morning!\n");
```

上面的程序段将打印无数次 Good morning!。为什么？因为循环中 index 的值一直都是原来的值 1，不曾变过。现在，考虑下面的程序段：

```
index = 1;
while (--index < 5)
    printf("Good morning!\n");
```

这段程序也好不到哪里去。虽然改变了 index 的值，但是改错了！不过，这个版本至少在 index 减

少到其类型到可容纳的最小负值并变成最大正值时会终止循环(第 3 章 3.4.2 节中的 toobig.c 程序解释过，最大正值加 1 一般会得到一个负值；类似地，最小负值减 1 一般会得到最大正值)。

6.2.2 何时终止循环

要明确一点：只有在对测试条件求值时，才决定是终止还是继续循环。例如，考虑程序清单 6.2 中的程序。

程序清单 6.2 when.c 程序

```
// when.c -- 何时退出循环
#include <stdio.h>
int main(void)
{
    int n = 5;

    while (n < 7)                    // 第 7 行
    {
        printf("n = %d\n", n);
        n++;                         // 第 10 行
        printf("Now n = %d\n", n);  // 第 11 行
    }
    printf("The loop has finished.\n");

    return 0;
}
```

运行程序清单 6.2，输出如下：

```
n = 5
Now n = 6
n = 6
Now n = 7
The loop has finished.
```

在第 2 次循环时，变量 n 在第 10 行首次获得值 7。但是，此时程序并未退出，它结束本次循环（第 11 行），并在对第 7 行的测试条件求值时才退出循环（变量 n 在第 1 次判断时为 5，第 2 次判断时为 6）。

6.2.3 while：入口条件循环

while 循环是使用入口条件的有条件循环。所谓“有条件”指的是语句部分的执行取决于测试表达式描述的条件，如(index < 5)。该表达式是一个入口条件（*entry condition*），因为必须满足条件才能进入循环体。在下面的情况中，就不会进入循环体，因为条件一开始就为假：

```
index = 10;
while (index++ < 5)
    printf("Have a fair day or better.\n");
```

把第 1 行改为：

```
index = 3;
```

就可以运行这个循环了。

6.2.4 语法要点

使用 while 时，要牢记一点：只有在测试条件后面的单独语句（简单语句或复合语句）才是循环部分。程序清单 6.3 演示了忽略这点的后果。缩进是为了让读者阅读方便，不是计算机的要求。

程序清单 6.3　while1.c **程序**

```
/* while1.c -- 注意花括号的使用 */
/* 糟糕的代码创建了一个无限循环 */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n < 3)
        printf("n is %d\n", n);
        n++;
    printf("That's all this program does\n");

    return 0;
}
```

该程序的输出如下：

```
n is 0
n is 0
n is 0
n is 0
n is 0
...
```

屏幕上会一直输出以上内容，除非强行关闭这个程序。

虽然程序中缩进了 `n++;`这条语句，但是并未把它和上一条语句括在花括号内。因此，只有直接跟在测试条件后面的一条语句是循环的一部分。变量 `n` 的值不会改变，条件 `n < 3` 一直为真。该循环会一直打印 `n is 0`，除非强行关闭程序。这是一个无限循环（*infinite loop*）的例子，没有外部干涉就不会退出。

记住，即使 `while` 语句本身使用复合语句，在语句构成上，它也是一条单独的语句。该语句从 `while` 开始执行，到第 1 个分号结束。在使用了复合语句的情况下，到右花括号结束。

要注意放置分号的位置。例如，考虑程序清单 6.4。

程序清单 6.4　while2.c **程序**

```
/* while2.c -- 注意分号的位置 */
#include <stdio.h>
int main(void)
{
    int n = 0;

    while (n++ < 3);            /* 第 7 行 */
       printf("n is %d\n", n); /* 第 8 行 */
    printf("That's all this program does.\n");

    return 0;
}
```

该程序的输出如下：

```
n is 4
That's all this program does.
```

如前所述，循环在执行完测试条件后面的第 1 条语句（简单语句或复合语句）后进入下一轮迭代，直到测试条件为假才会结束。该程序中第 7 行的测试条件后面直接跟着一个分号，循环在此进入下一轮迭代，因为单独一个分号被视为一条语句。虽然 n 的值在每次循环时都递增 1，但是第 8 行的语句不是循环的一部分，因此只会打印一次循环结束后的 n 值。

在该例中，测试条件后面的单独分号是*空语句*（*null statement*），它什么也不做。在 C 语言中，单独的分号表示空语句。有时，程序员会故意使用带空语句的 while 语句，因为所有的任务都在测试条件中完成了，不需要在循环体中做什么。例如，假设你想跳过输入到第 1 个非空白字符或数字，可以这样写：

```
while (scanf("%d", &num) == 1)
     ; /* 跳过整数输入 */
```

只要 scanf()读取一个整数，就会返回 1，循环继续执行。注意，为了提高代码的可读性，应该让这个分号独占一行，不要直接把它放在测试表达式同行。这样做一方面让读者更容易看到空语句，一方面也提醒自己和读者空语句是有意而为之。处理这种情况更好的方法是使用下一章介绍的 continue 语句。

6.3 用关系运算符和表达式比较大小

while 循环经常依赖测试表达式作比较，这样的表达式被称为*关系表达式*（*relational expression*），出现在关系表达式中间的运算符叫做*关系运算符*（*relational operator*）。前面的示例中已经用过一些关系运算符，表 6.1 列出了 C 语言的所有关系运算符。该表也涵盖了所有的数值关系（数字之间的关系再复杂也没有人与人之间的关系复杂）。

表 6.1 关系运算符

运算符	含义
<	小于
<=	小于或等于
==	等于
>=	大于或等于
>	大于
!=	不等于

关系运算符常用于构造 while 语句和其他 C 语句（稍后讨论）中用到的关系表达式。这些语句都会检查关系表达式为真还是为假。下面有 3 个互不相关的 while 语句，其中都包含关系表达式。

```
while (number < 6)
{
     printf("Your number is too small.\n");
     scanf("%d", &number);
}

while (ch != '$')
{
     count++;
     scanf("%c", &ch);
}

while (scanf("%f", &num) == 1)
     sum = sum + num;
```

注意，第 2 个 while 语句的关系表达式还可用于比较字符。比较时使用的是机器字符码（假定为 ASCII）。但是，不能用关系运算符比较字符串。第 11 章将介绍如何比较字符串。

虽然关系运算符也可用来比较浮点数，但是要注意：比较浮点数时，尽量只使用<和>。因为浮点数的舍入误差会导致在逻辑上应该相等的两数却不相等。例如，3 乘以 1/3 的积是 1.0。如果用把 1/3 表示成小数点后面 6 位数字，乘积则是.999999，不等于 1。使用 fabs() 函数（声明在 math.h 头文件中）可以方便地比较浮点数，该函数返回一个浮点值的绝对值（即，没有代数符号的值）。例如，可以用类似程序清单 6.5 的方法来判断一个数是否接近预期结果。

程序清单 6.5　cmpflt.c 程序

```
// cmpflt.c -- 浮点数比较
#include <math.h>
#include <stdio.h>
int main(void)
{
    const double ANSWER = 3.14159;
    double response;

    printf("What is the value of pi?\n");
    scanf("%lf", &response);
    while (fabs(response - ANSWER) > 0.0001)
    {
        printf("Try again!\n");
        scanf("%lf", &response);
    }
    printf("Close enough!\n");

    return 0;
}
```

循环会一直提示用户继续输入，除非用户输入的值与正确值之间相差 0.0001：

```
What is the value of pi?
3.14
Try again!
3.1416
Close enough!
```

6.3.1　什么是真

这是一个古老的问题，但是对 C 而言还不算难。在 C 中，表达式一定有一个值，关系表达式也不例外。程序清单 6.6 中的程序用于打印两个关系表达式的值，一个为真，一个为假。

程序清单 6.6　t_and_f.c 程序

```
/* t_and_f.c -- C 中的真和假的值 */
#include <stdio.h>
int main(void)
{
    int true_val, false_val;

    true_val = (10 > 2);        // 关系为真的值
    false_val = (10 == 2); // 关系为假的值
    printf("true = %d; false = %d \n", true_val, false_val);
```

```
    return 0;
}
```

程序清单 6.6 把两个关系表达式的值分别赋给两个变量，即把表达式为真的值赋给 true_val，表达式为假的值赋给 false_val。运行该程序后输出如下：

```
true = 1; false = 0
```

原来如此！对 C 而言，表达式为真的值是 1，表达式为假的值是 0。一些 C 程序使用下面的循环结构，由于 1 为真，所以循环会一直进行。

```
while (1)
{
    ...
}
```

6.3.2 其他真值

既然 1 或 0 可以作为 while 语句的测试表达式，是否还可以使用其他数字？如果可以，会发生什么？我们用程序清单 6.7 来做个实验。

程序清单 6.7　truth.c 程序

```
// truth.c -- 哪些值为真
#include <stdio.h>
int main(void)
{
    int n = 3;

    while (n)
        printf("%2d is true\n", n--);
    printf("%2d is false\n", n);

    n = -3;
    while (n)
        printf("%2d is true\n", n++);
    printf("%2d is false\n", n);

    return 0;
}
```

该程序的输出如下：

```
 3 is true
 2 is true
 1 is true
 0 is false
-3 is true
-2 is true
-1 is true
 0 is false
```

执行第 1 个循环时，n 分别是 3、2、1，当 n 等于 0 时，第 1 个循环结束。与此类似，执行第 2 个循环时，n 分别是-3、-2 和-1，当 n 等于 0 时，第 2 个循环结束。一般而言，所有的非零值都视为真，只有 0 被视为假。在 C 中，真的概念还真宽！

也可以说，只要测试条件的值为非零，就会执行 while 循环。这是从数值方面而不是从真/假方面来

看测试条件。要牢记：关系表达式为真，求值得 1；关系表达式为假，求值得 0。因此，这些表达式实际上相当于数值。

许多 C 程序员都会很好地利用测试条件的这一特性。例如，用 while (goats)替换 while (goats != 0)，因为表达式 goats != 0 和 goats 都只有在 goats 的值为 0 时才为 0 或假。第 1 种形式（while (goats != 0)）对初学者而言可能比较清楚，但是第 2 种形式（while (goats)）才是 C 程序员最常用的。要想成为一名 C 程序员，应该多熟悉 while (goats)这种形式。

6.3.3　真值的问题

C 对真的概念约束太少会带来一些麻烦。例如，我们稍微修改一下程序清单 6.1，修改后的程序如程序清单 6.8 所示。

程序清单 6.8　trouble.c **程序**

```
// trouble.c -- 误用=会导致无限循环

#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    int status;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    status = scanf("%ld", &num);
    while (status = 1)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        status = scanf("%ld", &num);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

运行该程序，其输出如下：

```
Please enter an integer to be summed (q to quit): 20
Please enter next integer (q to quit): 5
Please enter next integer (q to quit): 30
Please enter next integer (q to quit): q
Please enter next integer (q to quit):
Please enter next integer (q to quit):
Please enter next integer (q to quit):
Please enter next integer (q to quit):
```

（……屏幕上会一直显示最后的提示内容，除非强行关闭程序。也许你根本不想运行这个示例。）

这个麻烦的程序示例改动了 while 循环的测试条件，把 status == 1 替换成 status = 1。后者是一个赋值表达式语句，所以 status 的值为 1。而且，整个赋值表达式的值就是赋值运算符左侧的值，所以 status = 1 的值也是 1。这里，while (status = 1)实际上相当于 while (1)，也就是说，循环不会退出。虽然用户输入 q，status 被设置为 0，但是循环的测试条件把 status 又重置为 1，进入了

下一次迭代。

读者可能不太理解，程序的循环一直运行着，用户在输入 q 后完全没机会继续输入。如果 scanf()读取指定形式的输入失败，就把无法读取的输入留在输入队列中，供下次读取。当 scanf()把 q 作为整数读取时失败了，它把 q 留下。在下次循环时，scanf()从上次读取失败的地方（q）开始读取，scanf()把 q 作为整数读取，又失败了。因此，这样修改后不仅创建了一个无限循环，还创建了一个无限失败的循环，真让人沮丧。好在计算机觉察不出来。对计算机而言，无限地执行这些愚蠢的指令比成功预测未来 10 年的股市行情没什么两样。

不要在本应使用==的地方使用=。一些计算机语言（如，BASIC）用相同的符号表示赋值运算符和关系相等运算符，但是这两个运算符完全不同（见图 6.2）。赋值运算符把一个值赋给它左侧的变量；而关系相等运算符检查它左侧和右侧的值是否相等，不会改变左侧变量的值（如果左侧是一个变量）。

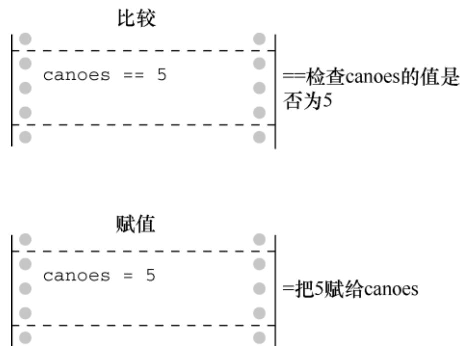


图 6.2　关系运算符==和赋值运算符=

示例如下：

```
canoes = 5          ←把 5 赋给 canoes
canoes == 5         ←检查 canoes 的值是否为 5
```

要注意使用正确的运算符。编译器不会检查出你使用了错误的形式，得出也不是预期的结果（误用=的人实在太多了，以至于现在大多数编译器都会给出警告，提醒用户是否要这样做）。如果待比较的一个值是常量，可以把该常量放在左侧有助于编译器捕获错误：

```
5 = canoes          ←语法错误
5 == canoes         ←检查 canoes 的值是否为 5
```

可以这样做是因为 C 语言不允许给常量赋值，编译器会把赋值运算符的这种用法作为语法错误标记出来。许多经验丰富的程序员在构建比较是否相等的表达式时，都习惯把常量放在左侧。

总之，关系运算符用于构成关系表达式。关系表达式为真时值为 1，为假时值为 0。通常用关系表达式作为测试条件的语句（如 while 和 if）可以使用任何表达式作为测试条件，非零为真，零为假。

6.3.4　新的_Bool 类型

在 C 语言中，一直用 int 类型的变量表示真/假值。C99 专门针对这种类型的变量新增了_Bool 类型。该类型是以英国数学家 George Boole 的名字命名的，他开发了用代数表示逻辑和解决逻辑问题。在编程中，表示真或假的变量被称为**布尔变量**（*Boolean variable*），所以_Bool 是 C 语言中布尔变量的类型名。_Bool 类型的变量只能储存 1（真）或 0（假）。如果把其他非零数值赋给_Bool 类型的变量，该变量会被设置为 1。这反映了 C 把所有的非零值都视为真。

程序清单 6.9 修改了程序清单 6.8 中的测试条件，把 int 类型的变量 status 替换为_Bool 类型的变量 input_is_good。给布尔变量取一个能表示真或假值的变量名是一种常见的做法。

程序清单 6.9　boolean.c 程序

```
// boolean.c -- 使用_Bool 类型的变量 variable
#include <stdio.h>
int main(void)
{
    long num;
    long sum = 0L;
    _Bool input_is_good;

    printf("Please enter an integer to be summed ");
    printf("(q to quit): ");
    input_is_good = (scanf("%ld", &num) == 1);
    while (input_is_good)
    {
        sum = sum + num;
        printf("Please enter next integer (q to quit): ");
        input_is_good = (scanf("%ld", &num) == 1);
    }
    printf("Those integers sum to %ld.\n", sum);

    return 0;
}
```

注意程序中把比较的结果赋值给_Bool 类型的变量 input_is_good：

```
input_is_good = (scanf("%ld", &num) == 1);
```

这样做没问题，因为==运算符返回的值不是 1 就是 0。顺带一提，从优先级方面考虑的话，并不需要用圆括号把 scanf("%ld", &num) == 1 括起来。但是，这样做可以提高代码可读性。还要注意，如何为变量命名才能让 while 循环的测试简单易懂：

```
while (input_is_good)
```

C99 提供了 stdbool.h 头文件，该头文件让 bool 成为_Bool 的别名，而且还把 true 和 false 分别定义为 1 和 0 的符号常量。包含该头文件后，写出的代码可以与 C++兼容，因为 C++把 bool、true 和 false 定义为关键字。

如果系统不支持_Bool 类型，导致无法运行该程序，可以把_Bool 替换成 int 即可。

6.3.5　优先级和关系运算符

关系运算符的优先级比算术运算符（包括+和-）低，比赋值运算符高。这意味着 x > y + 2 和 x > (y + 2)相同，x = y > 2 和 x = (y > 2)相同。换言之，如果 y 大于 2，则给 x 赋值 1，否则赋值 0。y 的值不会赋给 x。

关系运算符比赋值运算符的优先级高，因此，x_bigger = x > y;相当于 x_bigger = (x > y);。

关系运算符之间有两种不同的优先级。

高优先级组：　<<=　>>=

低优先级组：　==　!=

与其他大多数运算符一样，关系运算符的结合律也是从左往右。因此：

ex != wye == zee与(ex != wye) == zee相同

首先，C判断ex与wye是否相等；然后，用得出的值1或0（真或假）再与zee比较。我们并不推荐这样写，但是在这里有必要说明一下。

表6.2列出了目前我们学过的运算符的性质。附录B的参考资料II“C运算符”中列出了全部运算符的完整优先级表。

表6.2　运算符优先级

运算符（优先级从高至低）	结合律
()	从左往右
- + ++ -- sizeof	从右往左
* / %	从左往右
+ -	从左往右
< > <= >=	从左往右
== !=	从左往右
=	从右往左

小结：while语句

关键字： while

一般注解：

while语句创建了一个循环，重复执行直到测试表达式为假或0。while语句是一种入口条件循环，也就是说，在执行多次循环之前已决定是否执行循环。因此，循环体可以是简单语句，也可以是复合语句。

形式：

```
while ( expression )
     statement
```

在*expression*部分为假或0之前，重复执行*statement*部分。

示例：

```
while (n++ < 100)
    printf(" %d %d\n",n, 2 * n + 1); // 简单语句
while (fargo < 1000)
{ // 复合语句
    fargo = fargo + step;
    step = 2 * step;
}
```

小结：关系运算符和表达式

关系运算符：

每个关系运算符都把它左侧的值和右侧的值进行比较。

<	小于
<=	小于或等于

```
==          等于
>=          大于或等于
>           大于
!=          不等于
```

关系表达式：

简单的关系表达式由关系运算符及其运算对象组成。如果关系为真，关系表达式的值为 1；如果关系为假，关系表达式的值为 0。

示例：

```
5 > 2 为真，关系表达式的值为 1
(2 + a) == a 为假，关系表达式的值为 0
```

6.4　不确定循环和计数循环

一些 while 循环是不确定循环（*indefinite loop*）。所谓不确定循环，指在测试表达式为假之前，预先不知道要执行多少次循环。例如，程序清单 6.1 通过与用户交互获得数据来计算整数之和。我们事先并不知道用户会输入什么整数。另外，还有一类是计数循环（*counting loop*）。这类循环在执行循环之前就知道要重复执行多少次。程序清单 6.10 就是一个简单的计数循环。

程序清单 6.10　sweetie1.c 程序

```
// sweetie1.c -- 一个计数循环
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count = 1;                    // 初始化

    while (count <= NUMBER)           // 测试
    {
        printf("Be my Valentine!\n");   // 行为
        count++;                      // 更新计数
    }

    return 0;
}
```

虽然程序清单 6.10 运行情况良好，但是定义循环的行为并未组织在一起，程序的编排并不是很理想。我们来仔细分析一下。

在创建一个重复执行固定次数的循环中涉及了 3 个行为：

1．必须初始化计数器；
2．计数器与有限的值作比较；
3．每次循环时递增计数器。

while 循环的测试条件执行比较，递增运算符执行递增。程序清单 6.10 中，递增发生在循环的末尾，这可以防止不小心漏掉递增。因此，这样做比将测试和更新组合放在一起（即使用 count++ <= NUMBER）要好，但是计数器的初始化放在循环外，就有可能忘记初始化。实践告诉我们可能会发生的事情终究会发生，所以我们来学习另一种控制语句，可以避免这些问题。

6.5 for 循环

for 循环把上述 3 个行为（初始化、测试和更新）组合在一处。程序清单 6.11 使用 for 循环修改了程序清单 6.10 的程序。

程序清单 6.11 sweetie2.c 程序

```
// sweetie2.c -- 使用 for 循环的计数循环
#include <stdio.h>
int main(void)
{
    const int NUMBER = 22;
    int count;

    for (count = 1; count <= NUMBER; count++)
        printf("Be my Valentine!\n");

    return 0;
}
```

关键字 for 后面的圆括号中有 3 个表达式，分别用两个分号隔开。第 1 个表达式是初始化，只会在 for 循环开始时执行一次。第 2 个表达式是测试条件，在执行循环之前对表达式求值。如果表达式为假（本例中，count 大于 NUMBER 时），循环结束。第 3 个表达式执行更新，在每次循环结束时求值。程序清单 6.10 用这个表达式递增 count 的值，更新计数。完整的 for 语句还包括后面的简单语句或复合语句。for 圆括号中的表达式也叫做控制表达式，它们都是完整表达式，所以每个表达式的副作用（如，递增变量）都发生在对下一个表达式求值之前。图 6.3 演示了 for 循环的结构。

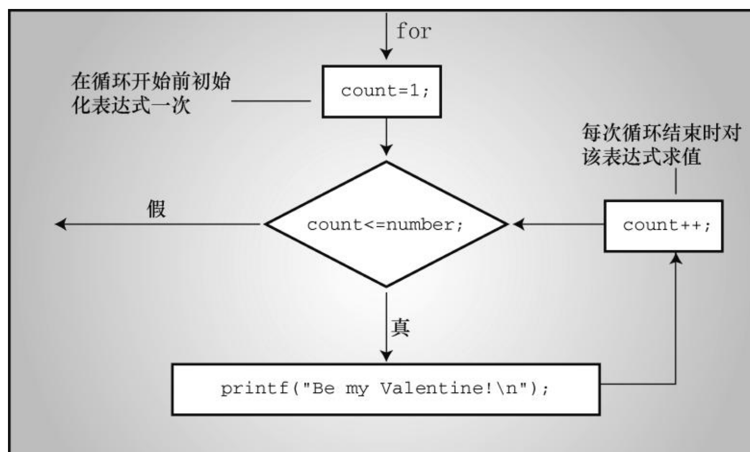


图 6.3 for 循环的结构

程序清单 6.12 for_cube.c 程序

```
/* for_cube.c -- 使用 for 循环创建一个立方表 */
#include <stdio.h>
int main(void)
{
    int num;

    printf("    n   n cubed\n");
```

```
    for (num = 1; num <= 6; num++)
        printf("%5d %5d\n", num, num*num*num);

    return 0;
}
```

程序清单 6.12 打印整数 1～6 及其对应的立方，该程序的输出如下：

```
n     n cubed
1         1
2         8
3        27
4        64
5       125
6       216
```

for 循环的第 1 行包含了循环所需的所有信息：num 的初值，num 的终值[1]和每次循环 num 的增量。

6.5.1　利用 for 的灵活性

虽然 for 循环看上去和 FORTRAN 的 DO 循环、Pascal 的 FOR 循环、BASIC 的 FOR...NEXT 循环类似，但是 for 循环比这些循环灵活。这些灵活性源于如何使用 for 循环中的 3 个表达式。以前面程序示例中的 for 循环为例，第 1 个表达式给计数器赋初值，第 2 个表达式表示计数器的范围，第 3 个表达式递增计数器。这样使用 for 循环确实很像其他语言的循环。除此之外，for 循环还有其他 9 种用法。

- 可以使用递减运算符来递减计数器：

```
/* for_down.c */
#include <stdio.h>
int main(void)
{
    int secs;

    for (secs = 5; secs > 0; secs--)
        printf("%d seconds!\n", secs);
    printf("We have ignition!\n");
    return 0;
}
```

该程序输出如下：

```
5 seconds!
4 seconds!
3 seconds!
2 seconds!
1 seconds!
We have ignition!
```

- 可以让计数器递增 2、10 等：

```
/* for_13s.c */
#include <stdio.h>
int main(void)
{
   int n; // 从 2 开始，每次递增 13
```

[1] 其实 num 的最终值不是 6，而是 7。虽然最后一次循环打印的 num 值是 6，但随后 num++使 num 的值为 7，然后 num <= 6 为假，for 循环结束。——译者注

```
    for (n = 2; n < 60; n = n + 13)
        printf("%d \n", n);
    return 0;
}
```

每次循环 n 递增 13，程序的输出如下：

```
 2
  15
28
41
54
```

■ 可以用字符代替数字计数：

```
/* for_char.c */
#include <stdio.h>
int main(void)
{
    char ch;

    for (ch = 'a'; ch <= 'z'; ch++)
        printf("The ASCII value for %c is %d.\n", ch, ch);
    return 0;
}
```

该程序假定系统用 ASCII 码表示字符。由于篇幅有限，省略了大部分输出：

```
The ASCII value for a is 97.
The ASCII value for b is 98.
...
The ASCII value for x is 120.
The ASCII value for y is 121.
The ASCII value for z is 122.
```

该程序能正常运行是因为字符在内部是以整数形式储存的，因此该循环实际上仍是用整数来计数。

■ 除了测试迭代次数外，还可以测试其他条件。在 for_cube 程序中，可以把：

```
for (num = 1; num <= 6; num++)
```

替换成：

```
for (num = 1; num*num*num <= 216; num++)
```

如果与控制循环次数相比，你更关心限制立方的大小，就可以使用这样的测试条件。

■ 可以让递增的量几何增长，而不是算术增长。也就是说，每次都乘上而不是加上一个固定的量：

```
/* for_geo.c */
#include <stdio.h>
int main(void)
{
    double debt;
    for (debt = 100.0; debt < 150.0; debt = debt * 1.1)
        printf("Your debt is now $%.2f.\n", debt);
    return 0;
}
```

该程序中，每次循环都把 debt 乘以 1.1，即 debt 的值每次都增加 10%，其输出如下：

```
Your debt is now $100.00.
Your debt is now $110.00.
Your debt is now $121.00.
Your debt is now $133.10.
Your debt is now $146.41.
```

■ 第 3 个表达式可以使用任意合法的表达式。无论是什么表达式，每次迭代都会更新该表达式的值。

```
/* for_wild.c */
#include <stdio.h>
int main(void)
{
    int x;
    int y = 55;

    for (x = 1; y <= 75; y = (++x * 5) + 50)
        printf("%10d %10d\n", x, y);
    return 0;
}
```

该循环打印 x 的值和表达式++x * 5 + 50 的值，程序的输出如下：

```
1        55
2        60
3        65
4        70
5        75
```

注意，测试涉及 y，而不是 x。for 循环中的 3 个表达式可以是不同的变量（注意，虽然该例可以正常运行，但是编程风格不太好。如果不在更新部分加入代数计算，程序会更加清楚）。

■ 可以省略一个或多个表达式（但是不能省略分号），只要在循环中包含能结束循环的语句即可。

```
/* for_none.c */
#include <stdio.h>
int main(void)
{
    int ans, n;
    ans = 2;
    for (n = 3; ans <= 25;)
        ans = ans * n;
    printf("n = %d; ans = %d.\n", n, ans);
    return 0;
}
```

该程序的输出如下：

```
n = 3; ans = 54.
```

该循环保持 n 的值为 3。变量 ans 开始的值为 2，然后递增到 6 和 18，最终是 54（18 比 25 小，所以 for 循环进入下一次迭代，18 乘以 3 得 54）。顺带一提，省略第 2 个表达式被视为真，所以下面的循环会一直运行：

```
for (; ; )
    printf("I want some action\n");
```

■ 第 1 个表达式不一定是给变量赋初值，也可以使用 printf()。记住，在执行循环的其他部分之前，只对第 1 个表达式求值一次或执行一次。

```
/* for_show.c */
#include <stdio.h>
int main(void)
{
    int num = 0;

    for (printf("Keep entering numbers!\n"); num != 6;)
        scanf("%d", &num);
    printf("That's the one I want!\n");
```

```
    return 0;
}
```

该程序打印第 1 行的句子一次，在用户输入 6 之前不断接受数字：

```
Keep entering numbers!
3
5
8
6
That's the one I want!
```

- 循环体中的行为可以改变循环头中的表达式。例如，假设创建了下面的循环：

  ```
  for (n = 1; n < 10000; n = n + delta)
  ```

  如果程序经过几次迭代后发现 delta 太小或太大，循环中的 if 语句（详见第 7 章）可以改变 delta 的大小。在交互式程序中，用户可以在循环运行时才改变 delta 的值。这样做也有危险的一面，例如，把 delta 设置为 0 就没用了。

总而言之，可以自己决定如何使用 for 循环头中的表达式，这使得在执行固定次数的循环外，还可以做更多的事情。接下来，我们将简要讨论一些运算符，使 for 循环更加有用。

小结：for 语句

关键字： for

一般注解：

for 语句使用 3 个表达式控制循环过程，分别用分号隔开。*initialize* 表达式在执行 for 语句之前只执行一次；然后对 *test* 表达式求值，如果表达式为真（或非零），执行循环一次；接着对 *update* 表达式求值，并再次检查 *test* 表达式。for 语句是一种入口条件循环，即在执行循环之前就决定了是否执行循环。因此，for 循环可能一次都不执行。*statement* 部分可以是一条简单语句或复合语句。

形式：

```
for ( initialize; test; update )
     statement
```

在 *test* 为假或 0 之前，重复执行 *statement* 部分。

示例：

```
for (n = 0; n < 10 ; n++)
    printf(" %d %d\n", n, 2 * n + 1);
```

6.6 其他赋值运算符：+=、-=、*=、/=、%=

C 有许多赋值运算符。最基本、最常用的是=，它把右侧表达式的值赋给左侧的变量。其他赋值运算符都用于更新变量，其用法都是左侧是一个变量名，右侧是一个表达式。赋给变量的新值是根据右侧表达式的值调整后的值。确切的调整方案取决于具体的运算符。例如：

```
scores += 20      与    scores = scores + 20      相同
dimes -= 2        与    dimes = dimes - 2         相同
bunnies *= 2      与    bunnies = bunnies * 2     相同
time /= 2.73      与    time = time / 2.73        相同
reduce %= 3       与    reduce = reduce % 3       相同
```

上述所列的运算符右侧都使用了简单的数，还可以使用更复杂的表达式，例如：

x *= 3 * y + 12 与 x = x * (3 * y + 12) 相同

以上提到的赋值运算符与=的优先级相同，即比+或*优先级低。上面最后一个例子也反映了赋值运算符的优先级，3 * y 先与 12 相加，再把计算结果与 x 相乘，最后再把乘积赋给 x。

并非一定要使用这些组合形式的赋值运算符。但是，它们让代码更紧凑，而且与一般形式相比，组合形式的赋值运算符生成的机器代码更高效。当需要在 for 循环中塞进一些复杂的表达式时，这些组合的赋值运算符特别有用。

6.7 逗号运算符

逗号运算符扩展了 for 循环的灵活性，以便在循环头中包含更多的表达式。例如，程序清单 6.13 演示了一个打印一类邮件资费（*first-class postage rate*）的程序（在撰写本书时，邮资为首重 40 美分/盎司，续重 20 美分/盎司，可以在互联网上查看当前邮资）。

程序清单 6.13 postage.c **程序**

```
// postage.c -- 一类邮资
#include <stdio.h>
int main(void)
{
    const int FIRST_OZ = 46;    // 2013 邮资
    const int NEXT_OZ = 20;     // 2013 邮资
    int ounces, cost;

    printf(" ounces  cost\n");
    for (ounces = 1, cost = FIRST_OZ; ounces <= 16; ounces++,cost += NEXT_OZ)
        printf("%5d   $%4.2f\n", ounces, cost / 100.0);

    return 0;
}
```

该程序的前 5 行输出如下：

```
ounces cost
1      $0.46
2      $0.66
3      $0.86
4      $1.06
```

该程序在初始化表达式和更新表达式中使用了逗号运算符。初始化表达式中的逗号使 ounces 和 cost 都进行了初始化，更新表达式中的逗号使每次迭代 ounces 递增 1、cost 递增 20（NEXT_Z 的值是 20）。绝大多数计算都在 for 循环头中进行（见图 6.4）。

逗号运算符并不局限于在 for 循环中使用，但是这是它最常用的地方。逗号运算符有两个其他性质。首先，它保证了被它分隔的表达式从左往右求值（换言之，逗号是一个序列点，所以逗号左侧项的所有副作用都在程序执行逗号右侧项之前发生）。因此，ounces 在 cost 之前被初始化。在该例中，顺序并不重要，但是如果 cost 的表达式中包含了 ounces 时，顺序就很重要。例如，假设有下面的表达式：

ounces++, cost = ounces * FIRST_OZ

在该表达式中，先递增 ounce，然后在第 2 个子表达式中使用 ounce 的新值。作为序列点的逗号保证了左侧子表达式的副作用在对右侧子表达式求值之前发生。

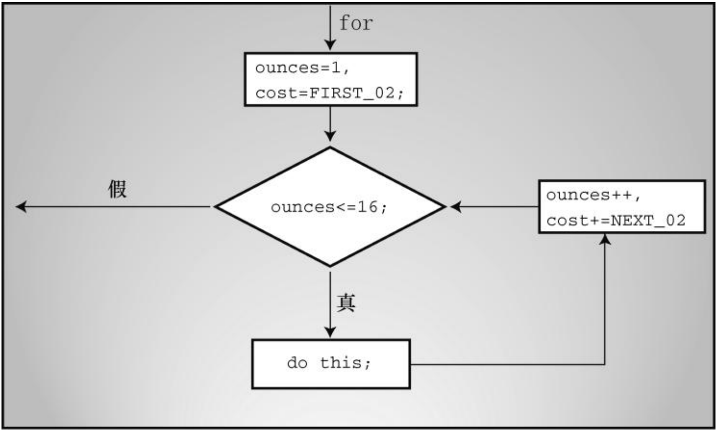


图 6.4 逗号运算符和 for 循环

其次，整个逗号表达式的值是右侧项的值。例如，下面语句

```
x = (y = 3, (z = ++y + 2) + 5);
```

的效果是：先把 3 赋给 y，递增 y 为 4，然后把 4 加 2 之和（6）赋给 z，接着加上 5，最后把结果 11 赋给 x。至于为什么有人编写这样的代码，在此不做评价。另一方面，假设在写数字时不小心输入了逗号：

```
houseprice = 249,500;
```

这不是语法错误，C 编译器会将其解释为一个逗号表达式，即 houseprice = 249 是逗号左侧的子表达式，500 是右侧的子表达式。因此，整个逗号表达式的值是逗号右侧表达式的值，而且左侧的赋值表达式把 249 赋给变量 houseprice。因此，这与下面代码的效果相同：

```
houseprice = 249;
500;
```

记住，任何表达式后面加上一个分号就成了表达式语句。所以，500;也是一条语句，但是什么也不做。

另外，下面的语句

```
houseprice = (249,500);
```

赋给 houseprice 的值是逗号右侧子表达式的值，即 500。

逗号也可用作分隔符。在下面语句中的逗号都是分隔符，不是逗号运算符：

```
char ch, date;
printf("%d %d\n", chimps, chumps);
```

小结：新的运算符

赋值运算符：

下面的运算符用右侧的值，根据指定的操作更新左侧的变量：

+=	把右侧的值加到左侧的变量上
-=	从左侧的变量中减去右侧的值
*=	把左侧的变量乘以右侧的值
/=	把左侧的变量除以右侧的值
%=	左侧变量除以右侧值得到的余数

示例：

rabbits *= 1.6;与 rabbits = rabbits * 1.6;相同

这些组合赋值运算符与普通赋值运算符的优先级相同，都比算术运算符的优先级低。因此，

contents *= old_rate + 1.2;

最终的效果与下面的语句相同：

contents = contents * (old_rate + 1.2);

逗号运算符：

逗号运算符把两个表达式连接成一个表达式，并保证最左边的表达式最先求值。逗号运算符通常在 for 循环头的表达式中用于包含更多的信息。整个逗号表达式的值是逗号右侧表达式的值。

示例：

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
    fargo += step;
```

6.7.1　当 Zeno 遇到 for 循环

接下来，我们看看 for 循环和逗号运算符如何解决古老的悖论。希腊哲学家 Zeno 曾经提出箭永远不会达到它的目标。首先，他认为箭要到达目标距离的一半，然后再达到剩余距离的一半，然后继续到达剩余距离的一半，这样就无穷无尽。Zeno 认为箭的飞行过程有无数个部分，所以要花费无数时间才能结束这一过程。不过，我们怀疑 Zeno 是自愿甘做靶子才会得出这样的结论。

我们采用一种定量的方法，假设箭用 1 秒钟走完一半的路程，然后用 1/2 秒走完剩余距离的一半，然后用 1/4 秒再走完剩余距离的一半，等等。可以用下面的无限序列来表示总时间：

```
1 + 1/2 + 1/4 + 1/8 + 1/16 +....
```

程序清单 6.14 中的程序求出了序列前几项的和。变量 power_of_two 的值分别是 1.0、2.0、4.0、8.0 等。

程序清单 6.14　zeno.c 程序

```
/* zeno.c -- 求序列的和 */
#include <stdio.h>

int main(void)
{
    int t_ct;       // 项计数
    double time, power_of_2;
    int limit;

    printf("Enter the number of terms you want: ");
    scanf("%d", &limit);
    for (time = 0, power_of_2 = 1, t_ct = 1; t_ct <= limit;
                                    t_ct++, power_of_2 *= 2.0)
    {
        time += 1.0 / power_of_2;
        printf("time = %f when terms = %d.\n", time, t_ct);
    }

    return 0;
}
```

下面是序列前 15 项的和：

```
Enter the number of terms you want: 15
 time = 1.000000 when terms = 1.
time = 1.500000 when terms = 2.
time = 1.750000 when terms = 3.
time = 1.875000 when terms = 4.
time = 1.937500 when terms = 5.
time = 1.968750 when terms = 6.
time = 1.984375 when terms = 7.
time = 1.992188 when terms = 8.
time = 1.996094 when terms = 9.
time = 1.998047 when terms = 10.
time = 1.999023 when terms = 11.
time = 1.999512 when terms = 12.
time = 1.999756 when terms = 13.
time = 1.999878 when terms = 14.
time = 1.999939 when terms = 15.
```

不难看出，尽管不断添加新的项，但是总和看起来变化不大。就像程序输出显示的那样，数学家的确证明了当项的数目接近无穷时，总和无限接近 2.0。假设 S 表示总和，下面我们用数学的方法来证明一下：

```
S = 1 + 1/2 + 1/4 + 1/8 + ...
```

这里的省略号表示“等等”。把 S 除以 2 得：

```
S/2 = 1/2 + 1/4 + 1/8 + 1/16 + ...
```

第 1 个式子减去第 2 个式子得：

```
S - S/2 = 1 +1/2 -1/2 + 1/4 -1/4 +...
```

除了第 1 个值为 1，其他的值都是一正一负地成对出现，所以这些项都可以消去。只留下：

```
S/2 = 1
```

然后，两侧同乘以 2，得：

```
S = 2
```

从这个示例中得到的启示是，在进行复杂的计算之前，先看看数学上是否有简单的方法可用。

程序本身是否有需要注意的地方？该程序演示了在表达式中可以使用多个逗号运算符，在 for 循环中，初始化了 time、power_of_2 和 count。构建完循环条件之后，程序本身就很简短了。

6.8　出口条件循环：do　while

while 循环和 for 循环都是入口条件循环，即在循环的每次迭代之前检查测试条件，所以有可能根本不执行循环体中的内容。C 语言还有出口条件循环（*exit-condition loop*），即在循环的每次迭代之后检查测试条件，这保证了至少执行循环体中的内容一次。这种循环被称为 do　while 循环。程序清单 6.15 演示了一个示例。

程序清单 6.15　do_while.c 程序

```
/* do_while.c -- 出口条件循环 */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    do
```

```
    {
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    } while (code_entered != secret_code);
    printf("Congratulations! You are cured!\n");

    return 0;
}
```

程序清单 6.15 在用户输入 13 之前不断提示用户输入数字。下面是一个运行示例：

```
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 12
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 14
To enter the triskaidekaphobia therapy club,
please enter the secret code number: 13
Congratulations! You are cured!
```

使用 `while` 循环也能写出等价的程序，但是长一些，如程序清单 6.16 所示。

程序清单 6.16　`entry.c` 程序

```
/* entry.c -- 出口条件循环 */
#include <stdio.h>
int main(void)
{
    const int secret_code = 13;
    int code_entered;

    printf("To enter the triskaidekaphobia therapy club,\n");
    printf("please enter the secret code number: ");
    scanf("%d", &code_entered);
    while (code_entered != secret_code)
    {
        printf("To enter the triskaidekaphobia therapy club,\n");
        printf("please enter the secret code number: ");
        scanf("%d", &code_entered);
    }
    printf("Congratulations! You are cured!\n");

    return 0;
}
```

下面是 `do while` 循环的通用形式：

```
do
    statement
while ( expression );
```

statement 可以是一条简单语句或复合语句。注意，`do while` 循环以分号结尾，其结构见图 6.5。

`do while` 循环在执行完循环体后才执行测试条件，所以至少执行循环体一次；而 `for` 循环或 `while` 循环都是在执行循环体之前先执行测试条件。`do while` 循环适用于那些至少要迭代一次的循环。例如，下面是一个包含 `do while` 循环的密码程序伪代码：

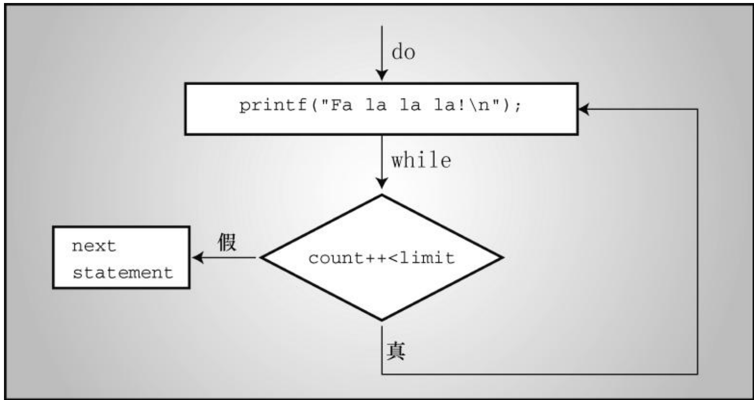


图 6.5　do　while 循环的结构

```
do
{
     提示用户输入密码
     读取用户输入的密码
} while (用户输入的密码不等于密码);
```

避免使用这种形式的 do　while 结构：

```
do
{
     询问用户是否继续
     其他行为
} while (回答是 yes);
```

这样的结构导致用户在回答“no”之后，仍然执行“其他行为”部分，因为测试条件执行晚了。

小结：do　while 语句

关键字：do　while

一般注解：

do　while 语句创建一个循环，在 *expression* 为假或 0 之前重复执行循环体中的内容。do　while 语句是一种出口条件循环，即在执行完循环体后才根据测试条件决定是否再次执行循环。因此，该循环至少必须执行一次。*statement* 部分可是一条简单语句或复合语句。

形式：

```
do
     statement
while ( expression );
```

在 *test* 为假或 0 之前，重复执行 *statement* 部分。

示例：

```
do
     scanf("%d", &number);
while (number != 20);
```

6.9　如何选择循环

如何选择使用哪一种循环？首先，确定是需要入口条件循环还是出口条件循环。通常，入口条件循环

用得比较多，有几个原因。其一，一般原则是在执行循环之前测试条件比较好。其二，测试放在循环的开头，程序的可读性更高。另外，在许多应用中，要求在一开始不满足测试条件时就直接跳过整个循环。

那么，假设需要一个入口条件循环，用 for 循环还是 while 循环？这取决于个人喜好，因为二者皆可。要让 for 循环看起来像 while 循环，可以省略第 1 个和第 3 个表达式。例如：

```
for ( ; test ; )
```

与下面的 while 效果相同：

```
while ( test )
```

要让 while 循环看起来像 for 循环，可以在 while 循环的前面初始化变量，并在 while 循环体中包含更新语句。例如：

```
初始化;
while ( 测试 )
{
     其他语句
     更新语句
}
```

与下面的 for 循环效果相同：

```
for ( 初始化 ;测试 ; 更新 )
     其他语句
```

一般而言，当循环涉及初始化和更新变量时，用 for 循环比较合适，而在其他情况下用 while 循环更好。对于下面这种条件，用 while 循环就很合适：

```
while (scanf("%ld", &num) == 1)
```

对于涉及索引计数的循环，用 for 循环更适合。例如：

```
for (count = 1; count <= 100; count++)
```

6.10　嵌套循环

嵌套循环（*nested loop*）指在一个循环内包含另一个循环。嵌套循环常用于按行和列显示数据，也就是说，一个循环处理一行中的所有列，另一个循环处理所有的行。程序清单 6.17 演示了一个简单的示例。

程序清单 6.17　rows1.c **程序**

```
/* rows1.c -- 使用嵌套循环 */
#include <stdio.h>
#define ROWS  6
#define CHARS 10
int main(void)
{
     int row;
     char ch;

     for (row = 0; row < ROWS; row++)                  /* 第 10 行 */
     {
          for (ch = 'A'; ch < ('A' + CHARS); ch++)     /* 第 12 行 */
               printf("%c", ch);
          printf("\n");
     }

     return 0;
}
```

运行该程序后，输出如下：

```
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
ABCDEFGHIJ
```

6.10.1　程序分析

第 10 行开始的 `for` 循环被称为*外层循环*（*outer loop*），第 12 行开始的 `for` 循环被称为*内层循环*（*inner loop*）。外层循环从 `row` 为 `0` 开始循环，到 `row` 为 `6` 时结束。因此，外层循环要执行 6 次，`row` 的值从 `0` 变为 `5`。每次迭代要执行的第 1 条语句是内层的 `for` 循环，该循环要执行 10 次，在同一行打印字符 `A`～`J`；第 2 条语句是外层循环的 `printf("\n");`，该语句的效果是另起一行，这样在下一次运行内层循环时，将在下一行打印的字符。

注意，嵌套循环中的内层循环在每次外层循环迭代时都执行完所有的循环。在程序清单 6.17 中，内层循环一行打印 10 个字符，外层循环创建 6 行。

6.10.2　嵌套变式

上一个实例中，内层循环和外层循环所做的事情相同。可以通过外层循环控制内层循环，在每次外层循环迭代时内层循环完成不同的任务。把程序清单 6.17 稍微修改后，如程序清单 6.18 所示。内层循环开始打印的字符取决于外层循环的迭代次数。该程序的第 1 行使用了新的注释风格，而且用 const 关键字代替 #define，有助于读者熟悉这两种方法。

程序清单 6.18　`rows2.c` 程序

```
// rows2.c -- 依赖外部循环的嵌套循环
#include <stdio.h>
int main(void)
{
    const int ROWS = 6;
    const int CHARS = 6;
    int row;
    char ch;

    for (row = 0; row < ROWS; row++)
    {
        for (ch = ('A' + row); ch < ('A' + CHARS); ch++)
            printf("%c", ch);
        printf("\n");
    }

    return 0;
}
```

该程序的输出如下：

```
ABCDEF
BCDEF
CDEF
DEF
EF
F
```

因为每次迭代都要把 row 的值与'A'相加，所以 ch 在每一行都被初始化为不同的字符。然而，测试条件并没有改变，所以每行依然是以 F 结尾，这使得每一行打印的字符都比上一行少一个。

6.11　数组简介

在许多程序中，数组很重要。数组可以作为一种储存多个相关项的便利方式。我们在第 10 章中将详细介绍数组，但是由于循环经常用到数组，所以在这里先简要地介绍一下。

数组（*array*）是按顺序储存的一系列类型相同的值，如 10 个 char 类型的字符或 15 个 int 类型的值。整个数组有一个数组名，通过整数下标访问数组中单独的项或元素（*element*）。例如，以下声明：

```
float debts[20];
```

声明 debts 是一个内含 20 个元素的数组，每个元素都可以储存 float 类型的值。数组的第 1 个元素是 debts[0]，第 2 个元素是 debts[1]，以此类推，直到 debts[19]。注意，数组元素的编号从 0 开始，不是从 1 开始。可以给每个元素赋 float 类型的值。例如，可以这样写：

```
debts[5] = 32.54;
debts[6] = 1.2e+21;
```

实际上，使用数组元素和使用同类型的变量一样。例如，可以这样把值读入指定的元素中：

```
scanf("%f", &debts[4]); // 把一个值读入数组的第 5 个元素
```

这里要注意一个潜在的陷阱：考虑到影响执行的速度，C 编译器不会检查数组的下标是否正确。下面的代码，都不正确：

```
debts[20] = 88.32;     // 该数组元素不存在
debts[33] = 828.12;    // 该数组元素不存在
```

编译器不会查找这样的错误。当运行程序时，这会导致数据被放置在已被其他数据占用的地方，可能会破坏程序的结果甚至导致程序异常中断。

数组的类型可以是任意数据类型。

```
int nannies[22];  /* 可储存 22 个 int 类型整数的数组 */
char actors[26];  /* 可储存 26 个字符的数组 */
long big[500];    /* 可储存 500 个 long 类型整数的数组 */
```

我们在第 4 章中讨论过字符串，可以把字符串储存在 char 类型的数组中（一般而言，char 类型数组的所有元素都储存 char 类型的值）。如果 char 类型的数组末尾包含一个表示字符串末尾的空字符\0，则该数组中的内容就构成了一个字符串（见图 6.6）。

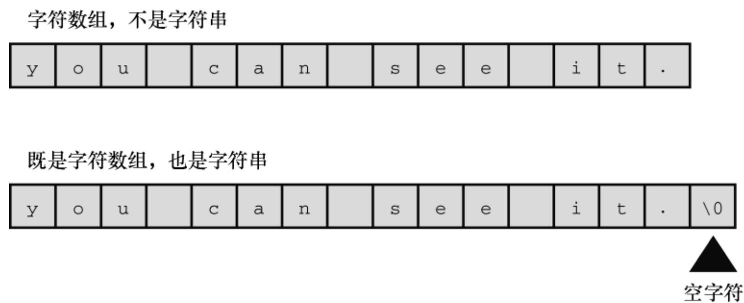


图 6.6　字符数组和字符串

用于识别数组元素的数字被称为下标（*subscript*）、索引（*indice*）或偏移量（*offset*）。下标必须是整数，

而且要从 0 开始计数。数组的元素被依次储存在内存中相邻的位置，如图 6.7 所示。

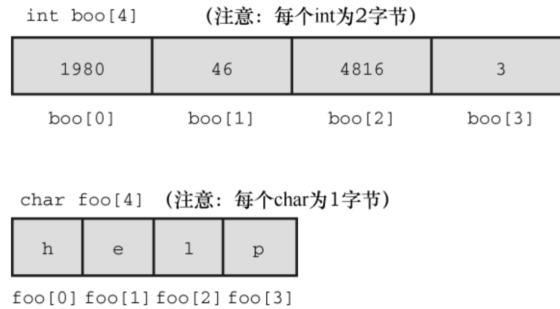


图 6.7　内存中的 char 和 int 类型的数组

6.11.1　在 for 循环中使用数组

程序中有许多地方要用到数组，程序清单 6.19 是一个较为简单的例子。该程序读取 10 个高尔夫分数，稍后进行处理。使用数组，就不用创建 10 个不同的变量来储存 10 个高尔夫分数。而且，还可以用 for 循环来读取数据。程序打印总分、平均分、差点（*handicap*，它是平均分与标准分的差值）。

程序清单 6.19　scores_in.c 程序

```
// scores_in.c -- 使用循环处理数组
#include <stdio.h>
#define SIZE 10
#define PAR 72
int main(void)
{
    int index, score[SIZE];
    int sum = 0;
    float average;

    printf("Enter %d golf scores:\n", SIZE);
    for (index = 0; index < SIZE; index++)
        scanf("%d", &score[index]);     // 读取 10 个分数
    printf("The scores read in are as follows:\n");
    for (index = 0; index < SIZE; index++)
        printf("%5d", score[index]);    // 验证输入
    printf("\n");
    for (index = 0; index < SIZE; index++)
        sum += score[index];            // 求总分数
    average = (float) sum / SIZE;       // 求平均分
    printf("Sum of scores = %d, average = %.2f\n", sum, average);
    printf("That's a handicap of %.0f.\n", average - PAR);

    return 0;
}
```

先看看程序清单 6.19 是否能正常工作，接下来再做一些解释。下面是程序的输出：

```
Enter 10 golf scores:
99 95 109 105 100
```

```
96 98 93 99 97 98
The scores read in are as follows:
  99 95 109 105 100 96 98 93 99 97
Sum of scores = 991, average = 99.10
That's a handicap of 27.
```

程序运行没问题，我们来仔细分析一下。首先，注意程序示例虽然打印了 11 个数字，但是只读入了 10 个数字，因为循环只读了 10 个值。由于 scanf()会跳过空白字符，所以可以在一行输入 10 个数字，也可以每行只输入一个数字，或者像本例这样混合使用空格和换行符隔开每个数字（因为输入是缓冲的，只有当用户键入 Enter 键后数字才会被发送给程序）。

然后，程序使用数组和循环处理数据，这比使用 10 个单独的 scanf()语句和 10 个单独的 printf()语句读取 10 个分数方便得多。for 循环提供了一个简单直接的方法来使用数组下标。注意，int 类型数组元素的用法与 int 类型变量的用法类似。要读取 int 类型变量 fue，应这样写：scanf("%d", &fue)。程序清单 6.19 中要读取 int 类型的元素 score[index]，所以这样写 scanf("%d", &score[index])。

该程序示例演示了一些较好的编程风格。第一，用#define 指令创建的明示常量（SIZE）来指定数组的大小。这样就可以在定义数组和设置循环边界时使用该明示常量。如果以后要扩展程序处理 20 个分数，只需简单地把 SIZE 重新定义为 20 即可，不用逐一修改程序中使用了数组大小的每一处。

第二，下面的代码可以很方便地处理一个大小为 SIZE 的数组：

```
for (index = 0; index < SIZE; index++)
```

设置正确的数组边界很重要。第 1 个元素的下标是 0，因此循环开始时把 index 设置为 0。因为从 0 开始编号，所以数组中最后一个元素的下标是 SIZE - 1。也就是说，第 10 个元素是 score[9]。通过测试条件 index < SIZE 来控制循环中使用的最后一个 index 的值是 SIZE - 1。

第三，程序能重复显示刚读入的数据。这是很好的编程习惯，有助于确保程序处理的数据与期望相符。

最后，注意该程序使用了 3 个独立的 for 循环。这是否必要？是否可以将其合并成一个循环？当然可以，读者可以动手试试，合并后的程序显得更加紧凑。但是，调整时要注意遵循模块化（*modularity*）的原则。模块化隐含的思想是：应该把程序划分为一些独立的单元，每个单元执行一个任务。这样做提高了程序的可读性。也许更重要的是，模块化使程序的不同部分彼此独立，方便后续更新或修改程序。在掌握如何使用函数后，可以把每个执行任务的单元放进函数中，提高程序的模块化。

6.12　使用函数返回值的循环示例

本章最后一个程序示例要用一个函数计算数的整数次幂（math.h 库提供了一个更强大幂函数 pow()，可以使用浮点指数）。该示例有 3 个主要任务：设计算法、在函数中表示算法并返回计算结果、提供一个测试函数的便利方法。

首先分析算法。为简化函数，我们规定该函数只处理正整数的幂。这样，把 n 与 n 相乘 p 次便可计算 n 的 p 次幂。这里自然会用到循环。先把变量 pow 设置为 1，然后将其反复乘以 n：

```
for(i = 1; i <= p; i++)
    pow *= n;
```

回忆一下，*=运算符把左侧的项乘以右侧的项，再把乘积赋给左侧的项。第 1 次循环后，pow 的值是 1 乘以 n，即 n；第 2 次循环后，pow 的值是上一次的值（n）乘以 n，即 n 的平方；以此类推。这种情况使用 for 循环很合适，因为在执行循环之前已预先知道了迭代的次数（已知 p）。

现在算法已确定，接下来要决定使用何种数据类型。指数 p 是整数，其类型应该是 int。为了扩大 n 及其幂的范围，n 和 pow 的类型都是 double。

接下来，考虑如何把以上内容用函数来实现。要使用两个参数（分别是 double 类型和 int 类型）才能把所需的信息传递给函数，并指定求哪个数的多少次幂。而且，函数要返回一个值。如何把函数的返回值返回给主调函数？编写一个有返回值的函数，要完成以下内容：

1．定义函数时，确定函数的返回类型；

2．使用关键字 return 表明待返回的值。

例如，可以这样写：

```
double power(double n, int p) // 返回一个 double 类型的值
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow; // 返回 pow 的值
}
```

要声明函数的返回类型，在函数名前写出类型即可，就像声明一个变量那样。关键字 return 表明该函数将把它后面的值返回给主调函数。根据上面的代码，函数返回一个变量的值。返回值也可以是表达式的值，如下所示：

```
return 2 * x + b;
```

函数将计算表达式的值，并返回该值。在主调函数中，可以把返回值赋给另一个变量、作为表达式中的值、作为另一个函数的参数（如，printf("%f", power(6.28, 3))），或者忽略它。

现在，我们在一个程序中使用这个函数。要测试一个函数很简单，只需给它提供几个值，看它是如何响应的。这种情况下可以创建一个输入循环，选择 while 循环很合适。可以使用 scanf()函数一次读取两个值。如果成功读取两个值，scanf()则返回 2，所以可以把 scanf()的返回值与 2 作比较来控制循环。还要注意，必须先声明 power()函数（即写出函数原型）才能在程序中使用它，就像先声明变量再使用一样。程序清单 6.20 演示了这个程序。

程序清单 6.20　powwer.c 程序

```
// power.c -- 计算数的整数幂
#include <stdio.h>
double power(double n, int p); // ANSI 函数原型
int main(void)
{
    double x, xpow;
    int exp;

    printf("Enter a number and the positive integer power");
    printf(" to which\nthe number will be raised. Enter q");
    printf(" to quit.\n");
    while (scanf("%lf%d", &x, &exp) == 2)
    {
        xpow = power(x, exp);  // 函数调用
        printf("%.3g to the power %d is %.5g\n", x, exp, xpow);
        printf("Enter next pair of numbers or q to quit.\n");
    }
    printf("Hope you enjoyed this power trip -- bye!\n");
```

```
    return 0;
}

double power(double n, int p)   // 函数定义
{
    double pow = 1;
    int i;

    for (i = 1; i <= p; i++)
        pow *= n;

    return pow;                 // 返回 pow 的值
}
```

运行该程序后，输出示例如下：

```
Enter a number and the positive integer power to which
the number will be raised. Enter q to quit.
1.2 12
1.2 to the power 12 is 8.9161
Enter next pair of numbers or q to quit.
2
16
2 to the power 16 is 65536
Enter next pair of numbers or q to quit.
q
Hope you enjoyed this power trip -- bye!
```

6.12.1 程序分析

该程序示例中的 main()是一个驱动程序（*driver*），即被设计用来测试函数的小程序。

该例的 while 循环是前面讨论过的一般形式。输入 1.2 12，scanf()成功读取两值，并返回 2，循环继续。因为 scanf()跳过空白，所以可以像输出示例那样，分多行输入。但是输入 q 会使 scanf()的返回值为 0，因为 q 与 scanf()中的转换说明%1f 不匹配。scanf()将返回 0，循环结束。类似地，输入 2.8 q 会使 scanf()的返回值为 1，循环也会结束。

现在分析一下与函数相关的内容。powwer()函数在程序中出现了 3 次。首次出现是：

```
double power(double n, int p); // ANSI 函数原型
```

这是 power()函数的原型，它声明程序将使用一个名为 power()的函数。开头的关键字 double 表明 power()函数返回一个 double 类型的值。编译器要知道 power()函数返回值的类型，才能知道有多少字节的数据，以及如何解释它们。这就是为什么必须声明函数的原因。圆括号中的 double n, int p 表示 power()函数的两个参数。第 1 个参数应该是 double 类型的值，第 2 个参数应该是 int 类型的值。

第 2 次出现是：

```
xpow = power(x,exp); // 函数调用
```

程序调用 power()，把两个值传递给它。该函数计算 x 的 exp 次幂，并把计算结果返回给主调函数。在主调函数中，返回值将被赋给变量 xpow。

第 3 次出现是：

```
double power(double n, int p) // 函数定义
```

这里，power()有两个形参，一个是 double 类型，一个是 int 类型，分别由变量 n 和变量 p 表示。

注意，函数定义的末尾没有分号，而函数原型的末尾有分号。在函数头后面花括号中的内容，就是 power() 完成任务的代码。

power()函数用 for 循环计算 n 的 p 次幂，并把计算结果赋给 pow，然后返回 pow 的值，如下所示：

```
return pow; //返回 pow 的值
```

6.12.2　使用带返回值的函数

声明函数、调用函数、定义函数、使用关键字 return，都是定义和使用带返回值函数的基本要素。

这里，读者可能有一些问题。例如，既然在使用函数返回值之前要声明函数，那么为什么在使用 scanf() 的返回值之前没有声明 scanf()？为什么在定义中说明了 power()的返回类型为 double，还要单独声明这个函数？

我们先回答第 2 个问题。编译器在程序中首次遇到 power()时，需要知道 power()的返回类型。此时，编译器尚未执行到 power()的定义，并不知道函数定义中的返回类型是 double。因此，必须通过前置声明（*forward declaration*）预先说明函数的返回类型。前置声明告诉编译器，power()定义在别处，其返回类型为 double。如果把 power()函数的定义置于 main()的文件顶部，就可以省略前置声明，因为编译器在执行到 main()之前已经知道 power()的所有信息。但是，这不是 C 的标准风格。因为 main() 通常只提供整个程序的框架，最好把 main()放在所有函数定义的前面。另外，通常把函数放在其他文件中，所以前置声明必不可少。

接下来，为什么不用声明 scanf()函数就可以使用它？其实，你已经声明了。stdio.h 头文件中包含了 scanf()、printf()和其他 I/O 函数的原型。scanf()函数的原型表明，它返回的类型是 int。

6.13　关键概念

循环是一个强大的编程工具。在创建循环时，要特别注意以下 3 个方面：

- 注意循环的测试条件要能使循环结束；
- 确保循环测试中的值在首次使用之前已初始化；
- 确保循环在每次迭代都更新测试的值。

C 通过求值来处理测试条件，结果为 0 表示假，非 0 表示真。带关系运算符的表达式常用于循环测试，它们有些特殊。如果关系表达式为真，其值为 1；如果为假，其值为 0。这与新类型_Bool 的值保持一致。

数组由相邻的内存位置组成，只储存相同类型的数据。记住，数组元素的编号从 0 开始，所有数组最后一个元素的下标一定比元素数目少 1。C 编译器不会检查数组下标值是否有效，自己要多留心。

使用函数涉及 3 个步骤：

- 通过函数原型声明函数；
- 在程序中通过函数调用使用函数；
- 定义函数。

函数原型是为了方便编译器查看程序中使用的函数是否正确，函数定义描述了函数如何工作。现代的编程习惯是把程序要素分为接口部分和实现部分，例如函数原型和函数定义。接口部分描述了如何使用一个特性，也就是函数原型所做的；实现部分描述了具体的行为，这正是函数定义所做的。

6.14　本章小结

本章的主题是程序控制。C 语言为实现结构化的程序提供了许多工具。while 语句和 for 语句提供了入口条件循环。for 语句特别适用于需要初始化和更新的循环。使用逗号运算符可以在 for 循环中初始化和更新多个变量。有些场合也需要使用出口条件循环，C 为此提供了 do while 语句。

典型的 while 循环设计的伪代码如下：

```
获得初值
while (值满足测试条件)
{
    处理该值
    获取下一个值
}
```

for 循环也可以完成相同的任务：

```
for (获得初值; 值满足测试条件; 获得下一个值)
    处理该值
```

这些循环都使用测试条件来判断是否继续执行下一次迭代。一般而言，如果对测试表达式求值为非 0，则继续执行循环；否则，结束循环。通常，测试条件都是关系表达式（由关系运算符和表达式构成）。表达式的关系为真，则表达式的值为 1；如果关系为假，则表达式的值为 0。C99 新增了_Bool 类型，该类型的变量只能储存 1 或 0，分别表示真或假。

除了关系运算符，本章还介绍了其他的组合赋值运算符，如+=或*=。这些运算符通过对其左侧运算对象执行算术运算来修改它的值。

接下来还简单地介绍了数组。声明数组时，方括号中的值指明了该数组的元素个数。数组的第 1 个元素编号为 0，第 2 个元素编号为 1，以此类推。例如，以下声明：

```
double hippos[20];
```

创建了一个有 20 个元素的数组 hippos，其元素从 hippos[0]～hippos[19]。利用循环可以很方便地操控数组的下标。

最后，本章演示了如何编写和使用带返回值的函数。

6.15　复习题

复习题的参考答案在附录 A 中。

1. 写出执行完下列各行后 quack 的值是多少。后 5 行中使用的是第 1 行 quack 的值。

```
int quack = 2;
quack += 5;
quack *= 10;
quack -= 6;
quack /= 8;
quack %= 3;
```

2. 假设 value 是 int 类型，下面循环的输出是什么？

```
for ( value = 36; value > 0; value /= 2)
      printf("%3d", value);
```

如果 value 是 double 类型，会出现什么问题？

3. 用代码表示以下测试条件：

   a. x大于5

   b. scanf()读取一个名为x的double类型值且失败

   c. x 的值等于5

4. 用代码表示以下测试条件：

   a. scanf()成功读入一个整数

   b. x 不等于5

   c. x 大于或等于20

5. 下面的程序有点问题，请找出问题所在。

```
#include <stdio.h>
int main(void)
{                                    /* 第 3 行 */
    int i, j, list(10);              /* 第 4 行 */

    for (i = 1, i <= 10, i++)        /* 第 6 行 */
    {                                /* 第 7 行 */
        list[i] = 2*i + 3;           /* 第 8 行 */
        for (j = 1, j > = i, j++)    /* 第 9 行 */
            printf(" %d", list[j]);  /* 第 10 行 */
        printf("\n");                /* 第 11 行 */
}                                    /* 第 12 行 */
```

6. 编写一个程序打印下面的图案，要求使用嵌套循环：

```
$$$$$$$$
$$$$$$$$
$$$$$$$$
$$$$$$$$
```

7. 下面的程序各打印什么内容？

a.

```
#include <stdio.h>

int main(void)
{
    int i = 0;

    while (++i < 4)
        printf("Hi! ");
    do
        printf("Bye! ");
    while (i++ < 8);
    return 0;
}
```

b.

```
#include <stdio.h>
int main(void)
{
    int i;
    char ch;
```

```
    for (i = 0, ch = 'A'; i < 4; i++, ch += 2 * i)
            printf("%c", ch);
    return 0;
}
```

8. 假设用户输入的是 `Go west, young man!`，下面各程序的输出是什么？（在 ASCII 码中，!紧跟在空格字符后面）

a.

```
#include <stdio.h>
int main(void)
{
    char ch;

    scanf("%c", &ch);
    while (ch != 'g')
    {
        printf("%c", ch);
        scanf("%c", &ch);
    }
    return 0;
}
```

b.

```
#include <stdio.h>

int main(void)
{
    char ch;

    scanf("%c", &ch);
    while (ch != 'g')
    {
        printf("%c", ++ch);
        scanf("%c", &ch);
    }
    return 0;
}
```

c.

```
#include <stdio.h>

int main(void)
{
    char ch;

    do {
        scanf("%c", &ch);
        printf("%c", ch);
    } while (ch != 'g');
    return 0;
}
```

d.

```
#include <stdio.h>
int main(void)
{
    char ch;
```

```
    scanf("%c", &ch);
    for (ch = '$'; ch != 'g'; scanf("%c", &ch))
        printf("%c", ch);
    return 0;
}
```

9．下面的程序打印什么内容？

```
#include <stdio.h>
int main(void)
{
    int n, m;

    n = 30;
    while (++n <= 33)
        printf("%d|", n);

    n = 30;
    do
        printf("%d|", n);
    while (++n <= 33);

    printf("\n***\n");

    for (n = 1; n*n < 200; n += 4)
        printf("%d\n", n);

    printf("\n***\n");

    for (n = 2, m = 6; n < m; n *= 2, m += 2)
        printf("%d %d\n", n, m);

    printf("\n***\n");

    for (n = 5; n > 0; n--)
    {
        for (m = 0; m <= n; m++)
            printf("=");
        printf("\n");
    }
    return 0;
}
```

10．考虑下面的声明：

```
double mint[10];
```

a．数组名是什么？

b．该数组有多少个元素？

c．每个元素可以储存什么类型的值？

d．下面的哪一个 `scanf()` 的用法正确？

i．`scanf("%lf", mint[2])`

ii．`scanf("%lf", &mint[2])`

iii．`scanf("%lf", &mint)`

11．Noah 先生喜欢以 2 计数，所以编写了下面的程序，创建了一个储存 2、4、6、8 等数字的数组。

这个程序是否有错误之处？如果有，请指出。

```
#include <stdio.h>
#define SIZE 8
int main(void)
{
    int by_twos[SIZE];
    int index;

    for (index = 1; index <= SIZE; index++)
         by_twos[index] = 2 * index;
    for (index = 1; index <= SIZE; index++)
         printf("%d ", by_twos);
    printf("\n");
    return 0;
}
```

12. 假设要编写一个返回 long 类型值的函数，函数定义中应包含什么？

13. 定义一个函数，接受一个 int 类型的参数，并以 long 类型返回参数的平方值。

14. 下面的程序打印什么内容？

```
#include <stdio.h>
int main(void)
{
    int k;
    for (k = 1, printf("%d: Hi!\n", k); printf("k = %d\n", k),
        k*k < 26; k += 2, printf("Now k is %d\n", k))
            printf("k is %d in the loop\n", k);
    return 0;
}
```

6.16　编程练习

1. 编写一个程序，创建一个包含 26 个元素的数组，并在其中储存 26 个小写字母。然后打印数组的所有内容。

2. 使用嵌套循环，按下面的格式打印字符：

```
$
$$
$$$
$$$$
$$$$$
```

3. 使用嵌套循环，按下面的格式打印字母：

```
F
FE
FED
FEDC
FEDCB
FEDCBA
```

注意：如果你的系统不使用 ASCII 或其他以数字顺序编码的代码，可以把字符数组初始化为字母表中的字母：

```
char lets[27] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

然后用数组下标选择单独的字母，例如 lets[0]是'A'，等等。

4．使用嵌套循环，按下面的格式打印字母：

```
A
BC
DEF
GHIJ
KLMNO
PQRSTU
```

如果你的系统不使用以数字顺序编码的代码，请参照练习 3 的方案解决。

5．编写一个程序，提示用户输入大写字母。使用嵌套循环以下面金字塔型的格式打印字母：

```
    A
   ABA
  ABCBA
 ABCDCBA
ABCDEDCBA
```

打印这样的图形，要根据用户输入的字母来决定。例如，上面的图形是在用户输入 E 后的打印结果。提示：用外层循环处理行，每行使用 3 个内层循环，分别处理空格、以升序打印字母、以降序打印字母。如果系统不使用 ASCII 或其他以数字顺序编码的代码，请参照练习 3 的解决方案。

6．编写一个程序打印一个表格，每一行打印一个整数、该数的平方、该数的立方。要求用户输入表格的上下限。使用一个 `for` 循环。

7．编写一个程序把一个单词读入一个字符数组中，然后倒序打印这个单词。提示：`strlen()`函数（第 4 章介绍过）可用于计算数组最后一个字符的下标。

8．编写一个程序，要求用户输入两个浮点数，并打印两数之差除以两数乘积的结果。在用户输入非数字之前，程序应循环处理用户输入的每对值。

9．修改练习 8，使用一个函数返回计算的结果。

10．编写一个程序，要求用户输入一个上限整数和一个下限整数，计算从上限到下限范围内所有整数的平方和，并显示计算结果。然后程序继续提示用户输入上限和下限整数，并显示结果，直到用户输入的上限整数小于下限整数为止。程序的运行示例如下：

```
Enter lower and upper integer limits: 5 9
The sums of the squares from 25 to 81 is 255
Enter next set of limits: 3 25
The sums of the squares from 9 to 625 is 5520
Enter next set of limits: 5 5
Done
```

11．编写一个程序，在数组中读入 8 个整数，然后按倒序打印这 8 个整数。

12．考虑下面两个无限序列：

```
1.0 + 1.0/2.0 + 1.0/3.0 + 1.0/4.0 + ...
1.0 - 1.0/2.0 + 1.0/3.0 - 1.0/4.0 + ...
```

编写一个程序计算这两个无限序列的总和，直到到达某次数。提示：奇数个-1 相乘得-1，偶数个-1 相乘得 1。让用户交互地输入指定的次数，当用户输入 0 或负值时结束输入。查看运行 100 项、1000 项、10000 项后的总和，是否发现每个序列都收敛于某值？

13．编写一个程序，创建一个包含 8 个元素的 `int` 类型数组，分别把数组元素设置为 2 的前 8 次幂。使用 `for` 循环设置数组元素的值，使用 `do while` 循环显示数组元素的值。

14．编写一个程序，创建两个包含 8 个元素的 double 类型数组，使用循环提示用户为第一个数组输入 8 个值。第二个数组元素的值设置为第一个数组对应元素的累积之和。例如，第二个数组的第 4

个元素的值是第一个数组前 4 个元素之和，第二个数组的第 5 个元素的值是第一个数组前 5 个元素之和（用嵌套循环可以完成，但是利用第二个数组的第 5 个元素是第二个数组的第 4 个元素与第一个数组的第 5 个元素之和，只用一个循环就能完成任务，不需要使用嵌套循环）。最后，使用循环显示两个数组的内容，第一个数组显示成一行，第二个数组显示在第一个数组的下一行，而且每个元素都与第一个数组各元素相对应。

15. 编写一个程序，读取一行输入，然后把输入的内容倒序打印出来。可以把输入储存在 char 类型的数组中，假设每行字符不超过 255。回忆一下，根据%c 转换说明，scanf()函数一次只能从输入中读取一个字符，而且在用户按下 Enter 键时 scanf()函数会生成一个换行字符（\n）。

16. Daphne 以 10%的单利息投资了 100 美元（也就是说，每年投资获利相当于原始投资的 10%）。Deirdre 以 5%的复合利息投资了 100 美元（也就是说，利息是当前余额的 5%，包含之前的利息）。编写一个程序，计算需要多少年 Deirdre 的投资额才会超过 Daphne，并显示那时两人的投资额。

17. Chuckie Lucky 赢得了 100 万美元（税后），他把奖金存入年利率 8%的账户。在每年的最后一天，Chuckie 取出 10 万美元。编写一个程序，计算多少年后 Chuckie 会取完账户的钱？

18. Rabnud 博士加入了一个社交圈。起初他有 5 个朋友。他注意到他的朋友数量以下面的方式增长。第 1 周少了 1 个朋友，剩下的朋友数量翻倍；第 2 周少了 2 个朋友，剩下的朋友数量翻倍。一般而言，第 N 周少了 N 个朋友，剩下的朋友数量翻倍。编写一个程序，计算并显示 Rabnud 博士每周的朋友数量。该程序一直运行，直到超过邓巴数（*Dunbar's number*）。邓巴数是粗略估算一个人在社交圈中有稳定关系的成员的最大值，该值大约是 150。

第 7 章

C 控制语句：分支和跳转

本章介绍以下内容：

- 关键字：if、else、switch、continue、break、case、default、goto
- 运算符：&&、||、?:
- 函数：getchar()、putchar()、ctype.h 系列
- 如何使用 if 和 if else 语句，如何嵌套它们
- 在更复杂的测试表达式中用逻辑运算符组合关系表达式
- C 的条件运算符
- switch 语句
- break、continue 和 goto 语句
- 使用 C 的字符 I/O 函数：getchar()和 putchar()
- ctype.h 头文件提供的字符分析函数系列

随着越来越熟悉 C，可以尝试用 C 程序解决一些更复杂的问题。这时候，需要一些方法来控制和组织程序，为此 C 提供了一些工具。前面已经学过如何在程序中用循环重复执行任务。本章将介绍分支结构（如，if 和 switch），让程序根据测试条件执行相应的行为。另外，还将介绍 C 语言的逻辑运算符，使用逻辑运算符能在 while 或 if 的条件中测试更多关系。此外，本章还将介绍跳转语句，它将程序流转换到程序的其他部分。学完本章后，读者就可以设计按自己期望方式运行的程序。

7.1 if 语句

我们从一个有 if 语句的简单示例开始学习，请看程序清单 7.1。该程序读取一列数据，每个数据都表示每日的最低温度（℃），然后打印统计的总天数和最低温度在 0℃以下的天数占总天数的百分比。程序中的循环通过 scanf()读入温度值。while 循环每迭代一次，就递增计数器增加天数，其中的 if 语句负责判断 0℃以下的温度并单独统计相应的天数。

程序清单 7.1 colddays.c 程序

```
// colddays.c -- 找出 0℃以下的天数占总天数的百分比
#include <stdio.h>
int main(void)
{
    const int FREEZING = 0;
    float temperature;
    int cold_days = 0;
    int all_days = 0;

    printf("Enter the list of daily low temperatures.\n");
```

```
    printf("Use Celsius, and enter q to quit.\n");
    while (scanf("%f", &temperature) == 1)
    {
        all_days++;
        if (temperature < FREEZING)
            cold_days++;
    }
    if (all_days != 0)
        printf("%d days total: %.1f%% were below freezing.\n",
             all_days, 100.0 * (float) cold_days / all_days);
    if (all_days == 0)
        printf("No data entered!\n");

    return 0;
}
```

下面是该程序的输出示例：

```
Enter the list of daily low temperatures.
Use Celsius, and enter q to quit.
12 5 -2.5 0 6 8 -3 -10 5 10 q
10 days total: 30.0% were below freezing.
```

while 循环的测试条件利用 scanf()的返回值来结束循环，因为 scanf()在读到非数字字符时会返回 0。temperature 的类型是 float 而不是 int，这样程序既可以接受-2.5 这样的值，也可以接受 8 这样的值。

while 循环中的新语句如下：

```
if (temperature < FREEZING)
    cold_days++;
```

if 语句指示计算机，**如果**刚读取的值（remperature）小于 0，就把 cold_days 递增 1；如果 temperature 不小于 0，就跳过 cold_days++;语句，while 循环继续读取下一个温度值。

接着，该程序又使用了两次 if 语句控制程序的输出。如果有数据，就打印结果；如果没有数据，就打印一条消息（稍后将介绍一种更好的方法来处理这种情况）。

为避免整数除法，该程序示例把计算后的百分比强制转换为 float 类型。其实，也不必使用强制类型转换，因为在表达式 100.0 * cold_days / all_days 中，将首先对表达式 100.0 * cold_days 求值，由于 C 的自动转换类型规则，乘积会被强制转换成浮点数。但是，使用强制类型转换可以明确表达转换类型的意图，保护程序免受不同版本编译器的影响。if 语句被称为**分支语句**（*branching statement*）或**选择语句**（*selection statement*），因为它相当于一个交叉点，程序要在两条分支中选择一条执行。if 语句的通用形式如下：

```
if ( expression )
    statement
```

如果对 *expression* 求值为真（非 0），则执行 *statement*；否则，跳过 *statement*。与 while 循环一样，*statement* 可以是一条简单语句或复合语句。if 语句的结构和 while 语句很相似，它们的主要区别是：如果满足条件可执行的话，if 语句只能测试和执行一次，而 while 语句可以测试和执行多次。

通常，*expression* 是关系表达式，即比较两个量的大小（如，表达式 x > y 或 c == 6）。如果 *expression* 为真（即 x 大于 y，或 c == 6），则执行 *statement*。否则，忽略 *statement*。概括地说，可以使用任意表达式，表达式的值为 0 则为假。

statement 部分可以是一条简单语句，如本例所示，或者是一条用花括号括起来的复合语句（或块）：

```
if (score > big)
    printf("Jackpot!\n");  // 简单语句

 if (joe > ron)
{                          // 复合语句
    joecash++;
    printf("You lose, Ron.\n");
}
```

注意，即使 if 语句由复合语句构成，整个 if 语句仍被视为一条语句。

7.2 if else 语句

简单形式的 if 语句可以让程序选择执行一条语句，或者跳过这条语句。C 还提供了 if else 形式，可以在两条语句之间作选择。我们用 if else 形式修正程序清单 7.1 中的程序段。

```
if (all_days != 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
if (all_days == 0)
    printf("No data entered!\n");
```

如果程序发现 all_days 不等于 0，那么它应该知道另一种情况一定是 all_days 等于 0。用 if else 形式只需测试一次。重写上面的程序段如下：

```
if (all_days!= 0)
    printf("%d days total: %.1f%% were below freezing.\n",
           all_days, 100.0 * (float) cold_days / all_days);
else
    printf("No data entered!\n");
```

如果 if 语句的测试表达式为真，就打印温度数据；如果为假，就打印警告消息。

注意，if else 语句的通用形式是：

```
if ( expression )
    statement1
else
    statement2
```

如果 *expression* 为真（非 0），则执行 *statement1*；如果 *expression* 为假或 0，则执行 else 后面的 *statement2*。*statement1* 和 *statement2* 可以是一条简单语句或复合语句。C 并不要求一定要缩进，但这是标准风格。缩进让根据测试条件的求值结果来判断执行哪部分语句一目了然。

如果要在 if 和 else 之间执行多条语句，必须用花括号把这些语句括起来成为一个块。下面的代码结构违反了 C 语法，因为在 if 和 else 之间只允许有一条语句（简单语句或复合语句）：

```
if (x > 0)
    printf("Incrementing x:\n");
    x++;
else      // 将产生一个错误
    printf("x <= 0 \n");
```

编译器把 printf()语句视为 if 语句的一部分，而把 x++;看作一条单独的语句，它不是 if 语句的一部分。然后，编译器发现 else 并没有所属的 if，这是错误的。上面的代码应该这样写：

```
if (x > 0)
{
    printf("Incrementing x:\n");
```

```
        x++;
    }
    else
        printf("x <= 0 \n");
```

if 语句用于选择是否执行一个行为，而 else if 语句用于在两个行为之间选择。图 7.1 比较了这两种语句。

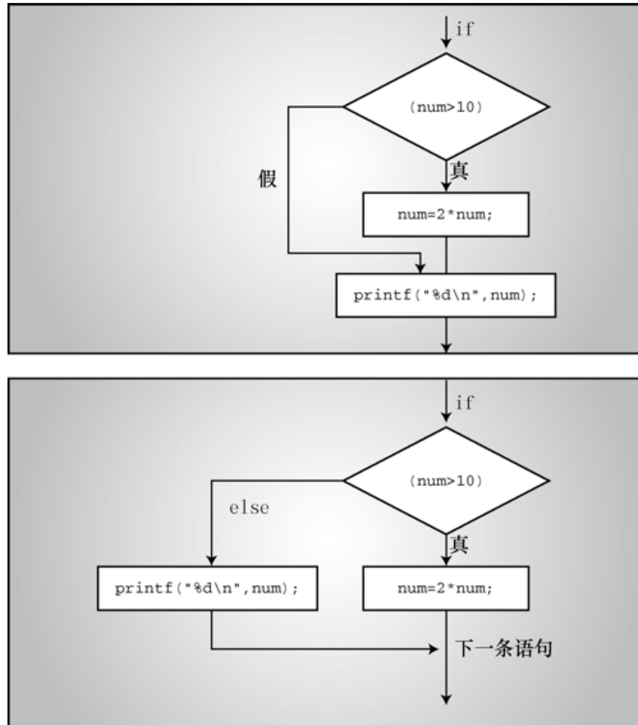


图 7.1　if 语句和 if else 语句

7.2.1　另一个示例：介绍 getchar()和 putchar()

到目前为止，学过的大多数程序示例都要求输入数值。接下来，我们看看输入字符的示例。相信读者已经熟悉了如何用 scanf()和 printf()根据%c 转换说明读写字符，我们马上要讲解的示例中要用到一对字符输入/输出函数：getchar()和 putchar()。

getchar()函数不带任何参数，它从输入队列中返回下一个字符。例如，下面的语句读取下一个字符输入，并把该字符的值赋给变量 ch：

```
ch = getchar();
```

该语句与下面的语句效果相同：

```
scanf("%c", &ch);
```

putchar()函数打印它的参数。例如，下面的语句把之前赋给 ch 的值作为字符打印出来：

```
putchar(ch);
```

该语句与下面的语句效果相同：

```
printf("%c", ch);
```

由于这些函数只处理字符，所以它们比更通用的 scanf()和 printf()函数更快、更简洁。而且，注意 getchar()和 putchar()不需要转换说明，因为它们只处理字符。这两个函数通常定义在 stdio.h

头文件中（而且，它们通常是预处理宏，而不是真正的函数，第 16 章会讨论类似函数的宏）。

接下来，我们编写一个程序来说明这两个函数是如何工作的。该程序把一行输入重新打印出来，但是每个非空格都被替换成原字符在 ASCII 序列中的下一个字符，空格不变。这一过程可描述为“如果字符是空白，原样打印；否则，打印原字符在 ASCII 序列中的下一个字符”。

C 代码看上去和上面的描述很相似，请看程序清单 7.2。

程序清单 7.2　cypher1.c 程序

```
// cypher1.c -- 更改输入，空格不变
#include <stdio.h>
#define SPACE ' '              // SPACE 表示单引号-空格-单引号
int main(void)
{
    char ch;

    ch = getchar();            // 读取一个字符
    while (ch != '\n')         // 当一行未结束时
    {
        if (ch == SPACE)       // 留下空格
            putchar(ch);       // 该字符不变
        else
            putchar(ch + 1);   // 改变其他字符
        ch = getchar();        // 获取下一个字符
    }
    putchar(ch);               // 打印换行符

    return 0;
}
```

（如果编译器警告因转换可能导致数据丢失，不用担心。第 8 章在讲到 EOF 时再解释 ）

下面是该程序的输入示例：

```
CALL ME HAL.
DBMM NF IBM/
```

把程序清单 7.1 中的循环和该例中的循环作比较。前者使用 scanf()返回的状态值判断是否结束循环，而后者使用输入项的值来判断是否结束循环。这使得两程序所用的循环结构略有不同：程序清单 7.1 中在循环前面有一条“读取语句”，程序清单 7.2 中在每次迭代的末尾有一条“读取语句”。不过，C 的语法比较灵活，读者也可以模仿程序清单 7.1，把读取和测试合并成一个表达式。也就是说，可以把这种形式的循环：

```
ch = getchar();        /* 读取一个字符 */
while (ch != '\n')     /* 当一行未结束时 */
{
    ...                /* 处理字符 */
    ch = getchar();    /* 获取下一个字符 */
}
```

替换成下面形式的循环：

```
while ((ch = getchar()) != '\n')
{
    ...                /* 处理字符 */
}
```

关键的一行代码是：

```
while ((ch = getchar()) != '\n')
```

这体现了 C 特有的编程风格——把两个行为合并成一个表达式。C 对代码的格式要求宽松，这样写让其中的每个行为更加清晰：

```
while (
        (ch = getchar())            // 给 ch 赋一个值
                          != '\n')  // 把 ch 和\n 作比较
```

以上执行的行为是赋值给 ch 和把 ch 的值与换行符作比较。表达式 ch = getchar()两侧的圆括号使之成为!=运算符的左侧运算对象。要对该表达式求值，必须先调用 getchar()函数，然后把该函数的返回值赋给 ch。因为赋值表达式的值是赋值运算符左侧运算对象的值，所以 ch = getchar()的值就是 ch 的新值，因此，读取 ch 的值后，测试条件相当于是 ch != '\n'（即，ch 不是换行符）。

这种独特的写法在 C 编程中很常见，应该多熟悉它。还要记住合理使用圆括号组合子表达式。上面例子中的圆括号都必不可少。假设省略 ch = getchar()两侧的圆括号：

```
while (ch = getchar() != '\n')
```

!=运算符的优先级比=高，所以先对表达式 getchar() != '\n'求值。由于这是关系表达式，所以其值不是 1 就是 0(真或假)。然后，把该值赋给 ch。省略圆括号意味着赋给 ch 的值是 0 或 1，而不是 getchar()的返回值。这不是我们的初衷。

下面的语句：

```
putchar(ch + 1); /* 改变其他字符 */
```

再次演示了字符实际上是作为整数储存的。为方便计算，表达式 ch + 1 中的 ch 被转换成 int 类型，然后 int 类型的计算结果被传递给接受一个 int 类型参数的 putchar()，该函数只根据最后一个字节确定显示哪个字符。

7.2.2　ctype.h 系列的字符函数

注意到程序清单 7.2 的输出中，最后输入的点号（.）被转换成斜杠（/），这是因为斜杠字符对应的 ASCII 码比点号的 ASCII 码多 1。如果程序只转换字母，保留所有的非字母字符（不只是空格）会更好。本章稍后讨论的逻辑运算符可用来测试字符是否不是空格、不是逗号等，但是列出所有的可能性太繁琐。C 有一系列专门处理字符的函数，ctype.h 头文件包含了这些函数的原型。这些函数接受一个字符作为参数，如果该字符属于某特殊的类别，就返回一个非零值（真）；否则，返回 0（假）。例如，如果 isalpha()函数的参数是一个字母，则返回一个非零值。程序清单 7.3 在程序清单 7.2 的基础上使用了这个函数，还使用了刚才精简后的循环。

程序清单 7.3　cypher2.c 程序

```
// cypher2.c -- 替换输入的字母，非字母字符保持不变
#include <stdio.h>
#include <ctype.h>            // 包含 isalpha()的函数原型
int main(void)
{
    char ch;

    while ((ch = getchar()) != '\n')
    {
        if (isalpha(ch))      // 如果是一个字符，
            putchar(ch + 1);  // 显示该字符的下一个字符
```

```
            else                    // 否则，
                putchar(ch);        // 原样显示
        }
        putchar(ch);                // 显示换行符

    return 0;
}
```

下面是该程序的一个输出示例，注意大小写字母都被替换了，除了空格和标点符号：

```
Look! It's a programmer!
Mppl! Ju't b qsphsbnnfs!
```

表 7.1 和表 7.2 列出了 ctype.h 头文件中的一些函数。有些函数涉及本地化，指的是为适应特定区域的使用习惯修改或扩展 C 基本用法的工具（例如，许多国家在书写小数点时，用逗号代替点号，于是特殊的本地化可以指定 C 编译器使用逗号以相同的方式输出浮点数，这样 123.45 可以显示为 123,45）。注意，字符映射函数不会修改原始的参数，这些函数只会返回已修改的值。也就是说，下面的语句不改变 ch 的值：

```
tolower(ch); // 不影响 ch 的值
```

这样做才会改变 ch 的值：

```
ch = tolower(ch); // 把 ch 转换成小写字母
```

表 7.1　ctype.h 头文件中的字符测试函数

函数名	如果是下列参数时，返回值为真
isalnum()	字母数字（字母或数字）
isalpha()	字母
isblank()	标准的空白字符（空格、水平制表符或换行符）或任何其他本地化指定为空白的字符
iscntrl()	控制字符，如 Ctrl+B
isdigit()	数字
isgraph()	除空格之外的任意可打印字符
islower()	小写字母
isprint()	可打印字符
ispunct()	标点符号（除空格或字母数字字符以外的任何可打印字符）
isspace()	空白字符（空格、换行符、换页符、回车符、垂直制表符、水平制表符或其他本地化定义的字符）
isupper()	大写字母
isxdigit()	十六进制数字符

表 7.2　ctype.h 头文件中的字符映射函数

函数名	行为
tolower()	如果参数是大写字符，该函数返回小写字符；否则，返回原始参数
toupper()	如果参数是小写字符，该函数返回大写字符；否则，返回原始参数

7.2.3　多重选择 else if

现实生活中我们经常有多种选择。在程序中也可以用 else if 扩展 if else 结构模拟这种情况。来看一个特殊的例子。电力公司通常根据客户的总用电量来决定电费。下面是某电力公司的电费清单，单位是千瓦时（kWh）：

```
首 360kWh:          $0.13230/kWh
续 108kWh:          $0.15040/kWh
续 252kWh:          $0.30025/kWh
超过 720kWh:        $0.34025/kWh
```

如果对用电管理感兴趣，可以编写一个计算电费的程序。程序清单 7.4 是完成这一任务的第 1 步。

程序清单 7.4　electric.c 程序

```
// electric.c -- 计算电费
#include <stdio.h>
#define RATE1   0.13230             // 首次使用 360 kwh 的费率
#define RATE2   0.15040             // 接着再使用 108 kwh 的费率
#define RATE3   0.30025             // 接着再使用 252 kwh 的费率
#define RATE4   0.34025             // 使用超过 720kwh 的费率
#define BREAK1  360.0               // 费率的第 1 个分界点
#define BREAK2  468.0               // 费率的第 2 个分界点
#define BREAK3  720.0               // 费率的第 3 个分界点
#define BASE1   (RATE1 * BREAK1)
// 使用360kwh 的费用
#define BASE2  (BASE1 + (RATE2 * (BREAK2 - BREAK1)))
// 使用468kwh 的费用
#define BASE3   (BASE1 + BASE2 + (RATE3 *(BREAK3 - BREAK2)))
// 使用720kwh 的费用
int main(void)
{
    double kwh;                     // 使用的千瓦时
    double bill;                    // 电费

    printf("Please enter the kwh used.\n");
    scanf("%lf", &kwh);             // %lf 对应 double 类型
    if (kwh <= BREAK1)
        bill = RATE1 * kwh;
    else if (kwh <= BREAK2)         // 360～468 kwh
        bill = BASE1 + (RATE2 * (kwh - BREAK1));
    else if (kwh <= BREAK3)         // 468～720 kwh
        bill = BASE2 + (RATE3 * (kwh - BREAK2));
    else                            // 超过 720 kwh
        bill = BASE3 + (RATE4 * (kwh - BREAK3));
    printf("The charge for %.1f kwh is $%1.2f.\n", kwh, bill);

    return 0;
}
```

该程序的输出示例如下：

```
Please enter the kwh used.
580
The charge for 580.0 kwh is $97.50.
```

程序清单 7.4 用符号常量表示不同的费率和费率分界点，以便把常量统一放在一处。这样，电力公司在更改费率以及费率分界点时，更新数据非常方便。BASE1 和 BASE2 根据费率和费率分界点来表示。一旦费率或分界点发生了变化，它们也会自动更新。预处理器是不进行计算的。程序中出现 BASE1 的地方都会被替换成 0.13230*360.0。不用担心，编译器会对该表达式求值得到一个数值（47.628），以便最终的程序代码使用的是 47.628 而不是一个计算式。

程序流简单明了。该程序根据 kwh 的值在 3 个公式之间选择一个。特别要注意的是，如果 kwh 大于或等于 360，程序只会到达第 1 个 else。因此，else　if (kwh <= BREAK2)这行相当于要求 kwh 在 360～482 之间，如程序注释所示。类似地，只有当 kwh 的值超过 720 时，才会执行最后的 else。最后，注意 BASE1、BASE2 和 BASE3 分别代表 360、468 和 720 千瓦时的总费用。因此，当电量超过这些值时，只需要加上额外的费用即可。

实际上，else　if 是已学过的 if　else 语句的变式。例如，该程序的核心部分只不过是下面代码的另一种写法：

```
if (kwh <= BREAK1)
    bill = RATE1 * kwh;
else
    if (kwh <= BREAK2)         // 360～468 kwh
        bill = BASE1 + (RATE2 * (kwh - BREAK1));
    else
        if (kwh <= BREAK3)     // 468～720 kwh
            bill = BASE2 + (RATE3 * (kwh - BREAK2));
        else                   // 超过 720 kwh
            bill = BASE3 + (RATE4 * (kwh - BREAK3));
```

也就是说，该程序由一个 ifelse 语句组成，else 部分包含另一个 if else 语句，该 if else 语句的 else 部分又包含另一个 if else 语句。第 2 个 if else 语句**嵌套**在第 1 个 if　else 语句中，第 3 个 if　else 语句嵌套在第 2 个 if　else 语句中。回忆一下，整个 if　else 语句被视为一条语句，因此不必把嵌套的 if　else 语句用花括号括起来。当然，花括号可以更清楚地表明这种特殊格式的含义。

这两种形式完全等价。唯一不同的是使用空格和换行的位置不同，不过编译器会忽略这些。尽管如此，第 1 种形式还是好些，因为这种形式更清楚地显示了有 4 种选择。在浏览程序时，这种形式让读者更容易看清楚各项选择。在需要时要缩进嵌套的部分，例如，必须测试两个单独的量时。本例中，仅在夏季对用电量超过 720kWh 的用户加收 10%的电费，就属于这种情况。

可以把多个 else　if 语句连成一串使用，如下所示（当然，要在编译器的限制范围内）：

```
if (score < 1000)
    bonus = 0;
else if (score < 1500)
    bonus = 1;
else if (score < 2000)
    bonus = 2;
else if (score < 2500)
    bonus = 4;
else
    bonus = 6;
```

（这可能是一个游戏程序的一部分，bonus 表示下一局游戏获得的光子炸弹或补给。）

对于编译器的限制范围，C99 标准要求编译器最少支持 127 层套嵌。

7.2.4　else 与 if 配对

如果程序中有许多 if 和 else，编译器如何知道哪个 if 对应哪个 else？例如，考虑下面的程序段：

```
if (number > 6)
    if (number < 12)
        printf("You're close!\n");
else
    printf("Sorry, you lose a turn!\n");
```

何时打印 Sorry, you lose a turn!？当 number 小于或等于 6 时，还是 number 大于 12 时？换言之，else 与第 1 个 if 还是第 2 个 if 匹配？答案是，else 与第 2 个 if 匹配。也就是说，输入的数字和匹配的响应如下：

数字	响应
5	None
10	You’re close!
15	Sorry, you lose a turn!

规则是，如果没有花括号，else 与离它最近的 if 匹配，除非最近的 if 被花括号括起来（见图 7.2）。

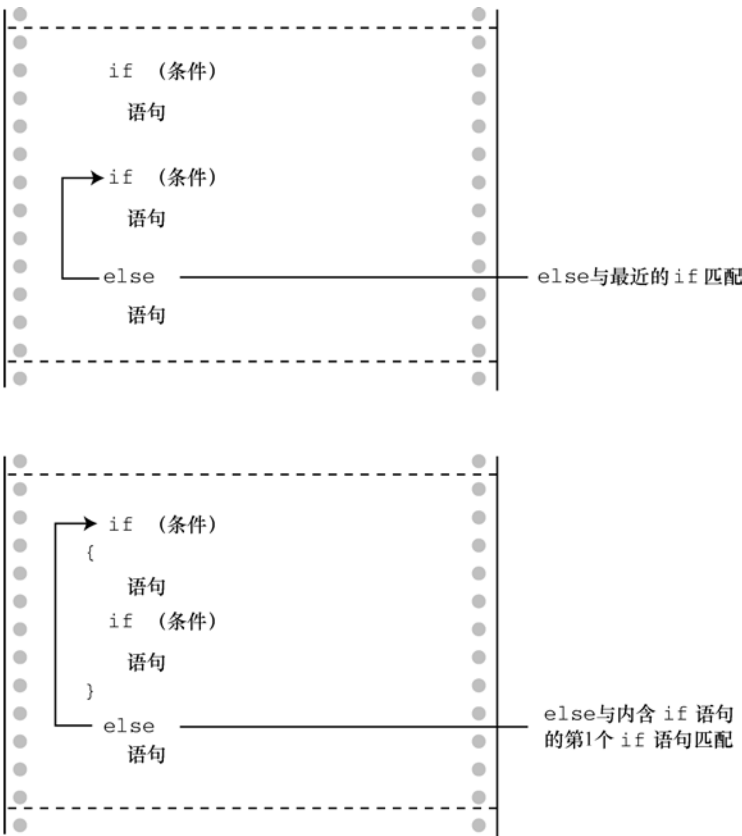


图 7.2　if　else 匹配的规则

注意：要缩进“语句”，“语句”可以是一条简单语句或复合语句。

第 1 个例子的缩进使得 else 看上去与第 1 个 if 相匹配，但是记住，编译器是忽略缩进的。如果希望 else 与第 1 个 if 匹配，应该这样写：

```
 if (number > 6)
{
    if (number < 12)
        printf("You're close!\n");
}
else
    printf("Sorry, you lose a turn!\n");
```

这样改动后，响应如下：

数字	响应
5	Sorry, you lose a turn!
10	You’re close!
15	None

7.2.5　多层嵌套的if语句

前面介绍的if...else　if...else序列是嵌套if的一种形式，从一系列选项中选择一个执行。有时，选择一个特定选项后又引出其他选择，这种情况可以使用另一种嵌套if。例如，程序可以使用if　else选择男女，if　else的每个分支里又包含另一个if　else来区分不同收入的群体。

我们把这种形式的嵌套if应用在下面的程序中。给定一个整数，显示所有能整除它的约数。如果没有约数，则报告该数是一个素数。

在编写程序的代码之前要先规划好。首先，要总体设计一下程序。为方便起见，程序应该使用一个循环让用户能连续输入待测试的数。这样，测试一个新的数字时不必每次都要重新运行程序。下面是我们为这种循环开发的一个模型（伪代码）：

```
提示用户输入数字
当scanf()返回值为1
    分析该数并报告结果
    提示用户继续输入
```

回忆一下在测试条件中使用scanf()，把读取数字和判断测试条件确定是否结束循环合并在一起。

下一步，设计如何找出约数。也许最直接的方法是：

```
for (div = 2; div < num; div++)
    if (num % div == 0)
        printf("%d is divisible by %d\n", num, div);
```

该循环检查2～num之间的所有数字，测试它们是否能被num整除。但是，这个方法有点浪费时间。我们可以改进一下。例如，考虑如果144%2得0，说明2是144的约数；如果144除以2得72，那么72也是144的一个约数。所以，num % div测试成功可以获得两个约数。为了弄清其中的原理，我们分析一下循环中得到的成对约数：2和72、2和48、4和36、6和24、8和18、9和16、12和12、16和9、18和8，等等。在得到12和12这对约数后，又开始得到已找到的相同约数（次序相反）。因此，不用循环到143，在达到12以后就可以停止循环。这大大地节省了循环时间！

分析后发现，必须测试的数只要到num的平方根就可以了，不用到num。对于9这样的数字，不会节约很多时间，但是对于10000这样的数，使用哪一种方法求约数差别很大。不过，我们不用在程序中计算平方根，可以这样编写测试条件：

```
for (div = 2; (div * div) <= num; div++)
    if (num % div == 0)
        printf("%d is divisible by %d and %d.\n",num, div, num / div);
```

如果 num 是 144，当 div = 12 时停止循环。如果 num 是 145，当 div = 13 时停止循环。

不使用平方根而用这样的测试条件，有两个原因。其一，整数乘法比求平方根快。其二，我们还没有正式介绍平方根函数。

还要解决两个问题才能准备编程。第 1 个问题，如果待测试的数是一个完全平方数怎么办？报告 144 可以被 12 和 12 整除显得有点傻。可以使用嵌套 if 语句测试 div 是否等于 num /div。如果是，程序只打印一个约数：

```
for (div = 2; (div * div) <= num; div++)
{
    if (num % div == 0)
    {
        if (div * div != num)
            printf("%d is divisible by %d and %d.\n",num, div, num / div);
        else
            printf("%d is divisible by %d.\n", num, div);
    }
}
```

注意

从技术角度看，if else 语句作为一条单独的语句，不必使用花括号。外层 if 也是一条单独的语句，也不必使用花括号。但是，当语句太长时，使用花括号能提高代码的可读性，而且还可防止今后在 if 循环中添加其他语句时忘记加花括号。

第 2 个问题，如何知道一个数字是素数？如果 num 是素数，程序流不会进入 if 语句。要解决这个问题，可以在外层循环把一个变量设置为某个值（如，1），然后在 if 语句中把该变量重新设置为 0。循环完成后，检查该变量是否是 1，如果是，说明没有进入 if 语句，那么该数就是素数。这样的变量通常称为**标记**（*flag*）。

一直以来，C 都习惯用 int 作为标记的类型，其实新增的_Bool 类型更合适。另外，如果在程序中包含了 stdbool.h 头文件，便可用 bool 代替_Bool 类型，用 true 和 false 分别代替 1 和 0。

程序清单 7.5 体现了以上分析的思路。为扩大该程序的应用范围，程序用 long 类型而不是 int 类型（如果系统不支持_Bool 类型，可以把 isPrime 的类型改为 int，并用 1 和 0 分别替换程序中的 true 和 false）。

程序清单 7.5　divisors.c 程序

```
// divisors.c -- 使用嵌套 if 语句显示一个数的约数
#include <stdio.h>
#include <stdbool.h>
int main(void)
{
    unsigned long num;          // 待测试的数
    unsigned long div;          // 可能的约数
    bool isPrime;               // 素数标记

    printf("Please enter an integer for analysis; ");
    printf("Enter q to quit.\n");
    while (scanf("%lu", &num) == 1)
    {
```

```
        for (div = 2, isPrime = true; (div * div) <= num; div++)
        {
            if (num % div == 0)
             {
                if ((div * div) != num)
                printf("%lu is divisible by %lu and %lu.\n",
                        num, div, num / div);
                else
                    printf("%lu is divisible by %lu.\n",
                        num, div);
                isPrime = false;   // 该数不是素数
            }
        }
        if (isPrime)
            printf("%lu is prime.\n", num);
        printf("Please enter another integer for analysis; ");
        printf("Enter q to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}
```

注意，该程序在 `for` 循环的测试表达式中使用了逗号运算符，这样每次输入新值时都可以把 `isPrime` 设置为 `true`。

下面是该程序的一个输出示例：

```
Please enter an integer for analysis; Enter q to quit.
123456789
123456789 is divisible by 3 and 41152263.
123456789 is divisible by 9 and 13717421.
123456789 is divisible by 3607 and 34227.
123456789 is divisible by 3803 and 32463.
123456789 is divisible by 10821 and 11409.
Please enter another integer for analysis; Enter q to quit.
149
149 is prime.
Please enter another integer for analysis; Enter q to quit.
2013
2013 is divisible by 3 and 671.
2013 is divisible by 11 and 183.
2013 is divisible by 33 and 61.
Please enter another integer for analysis; Enter q to quit.
q
Bye.
```

该程序会把 1 认为是素数，其实它不是。下一节将要介绍的逻辑运算符可以排除这种特殊的情况。

小结：用 if 语句进行选择

关键字：if、else

一般注解：

下面各形式中，*statement* 可以是一条简单语句或复合语句。表达式为真说明其值是非零值。

形式 1：

```
if (expression)
    statement
```

如果 *expression* 为真，则执行 *statement* 部分。

形式 2：

```
if (expression)
    statement1
else
    statement2
```

如果 *expression* 为真，执行 *statement1* 部分；否则，执行 *statement2* 部分。

形式 3：

```
if (expression1)
    statement1
else if (expression2)
    statement2
else
    statement3
```

如果 *expression1* 为真，执行 *statement1* 部分；如果 *expression2* 为真，执行 *statement2* 部分；否则，执行 *statement3* 部分。

示例：

```
if (legs == 4)
    printf("It might be a horse.\n");
else if (legs > 4)
    printf("It is not a horse.\n");
else     /* 如果 legs < 4 */
{
    legs++;
    printf("Now it has one more leg.\n");
}
```

7.3　逻辑运算符

读者已经很熟悉了，if 语句和 while 语句通常使用关系表达式作为测试条件。有时，把多个关系表达式组合起来会很有用。例如，要编写一个程序，计算输入的一行句子中除单引号和双引号以外其他字符的数量。这种情况下可以使用逻辑运算符，并使用句点（.）标识句子的末尾。程序清单 7.6 用一个简短的程序进行演示。

程序清单 7.6　chcount.c 程序

```
// chcount.c  -- 使用逻辑与运算符
#include <stdio.h>
#define PERIOD '.'
int main(void)
{
    char ch;
    int charcount = 0;

    while ((ch = getchar()) != PERIOD)
```

```
    {
        if (ch != '"' && ch != '\'')
            charcount++;
    }
    printf("There are %d non-quote characters.\n", charcount);

    return 0;
}
```

下面是该程序的一个输出示例：

```
I didn't read the "I'm a Programming Fool" best seller.
There are 50 non-quote characters.
```

程序首先读入一个字符，并检查它是否是一个句点，因为句点标志一个句子的结束。接下来，if 语句的测试条件中使用了**逻辑与**运算符&&。该 if 语句翻译成文字是“如果待测试的字符不是双引号，并且它也不是单引号，那么 charcount 递增 1”。

逻辑运算符两侧的条件必须都为真，整个表达式才为真。逻辑运算符的优先级比关系运算符低，所以不必在子表达式两侧加圆括号。

C 有 3 种逻辑运算符，见表 7.3。

表 7.3 种逻辑运算符

逻辑运算符	含义
&&	与
||	或
!	非

假设 exp1 和 exp2 是两个简单的关系表达式（如 car > rat 或 debt == 1000），那么：

- 当且仅当 exp1 和 exp2 都为真时，exp1 && exp2 才为真；
- 如果 exp1 或 exp2 为真，则 exp1 || exp2 为真；
- 如果 exp1 为假，则!exp1 为真；如果 exp1 为真，则!exp1 为假。

下面是一些具体的例子：

5 > 2 && 4 > 7 为假，因为只有一个子表达式为真；
5 > 2 || 4 > 7 为真，因为有一个子表达式为真；
!(4 > 7)为真，因为 4 不大于 7。

顺带一提，最后一个表达式与下面的表达式等价：

```
4 <= 7
```

如果不熟悉逻辑运算符或者觉得很别扭，请记住：(练习&&时间)== 完美。

7.3.1 备选拼写：iso646.h 头文件

C 是在美国用标准美式键盘开发的语言。但是在世界各地，并非所有的键盘都有和美式键盘一样的符号。因此，C99 标准新增了可代替逻辑运算符的拼写，它们被定义在 ios646.h 头文件中。如果在程序中包含该头文件，便可用 and 代替&&、or 代替||、not 代替!。例如，可以把下面的代码：

```
if (ch != '"' && ch != '\'')
    charcount++;
```

改写为：

```
if (ch != '"' and ch != '\'')
    charcount++;
```

表 7.4 列出了逻辑运算符对应的拼写，很容易记。读者也许很好奇，为何 C 不直接使用 and、or 和 not？因为 C 一直坚持尽量保持较少的关键字。参考资料 V“新增 C99 和 C11 的标准 ANSI C 库”列出了一些运算符的备选拼写，有些我们还没见过。

表 7.4　逻辑运算符的备选拼写

传统写法	iso646.h
&&	and
||	or
!	not

7.3.2　优先级

!运算符的优先级很高，比乘法运算符还高，与递增运算符的优先级相同，只比圆括号的优先级低。&&运算符的优先级比||运算符高，但是两者的优先级都比关系运算符低，比赋值运算符高。因此，表达式 a > b && b > c || b > d 相当于((a > b) && (b > c)) || (b > d)。

也就是说，b 介于 a 和 c 之间，或者 b 大于 d。

尽管对于该例没必要使用圆括号，但是许多程序员更喜欢使用带圆括号的第 2 种写法。这样做即使不记得逻辑运算符的优先级，表达式的含义也很清楚。

7.3.3　求值顺序

除了两个运算符共享一个运算对象的情况外，C 通常不保证先对复杂表达式中哪部分求值。例如，下面的语句，可能先对表达式 5 + 3 求值，也可能先对表达式 9 + 6 求值：

```
apples = (5 + 3) * (9 + 6);
```

C 把先计算哪部分的决定权留给编译器的设计者，以便针对特定系统优化设计。但是，对于逻辑运算符是个例外，C 保证逻辑表达式的求值顺序是从左往右。&&和||运算符都是序列点，所以程序在从一个运算对象执行到下一个运算对象之前，所有的副作用都会生效。而且，C 保证一旦发现某个元素让整个表达式无效，便立即停止求值。正是由于有这些规定，才能写出这样结构的代码：

```
while ((c = getchar()) != ' ' && c != '\n')
```

如上代码所示，读取字符直至遇到第 1 个空格或换行符。第 1 个子表达式把读取的值赋给 c，后面的子表达式会用到 c 的值。如果没有求值循序的保证，编译器可能在给 c 赋值之前先对后面的表达式求值。

这里还有一个例子：

```
if (number != 0 && 12/number == 2)
    printf("The number is 5 or 6.\n");
```

如果 number 的值是 0，那么第 1 个子表达式为假，且不再对关系表达式求值。这样避免了把 0 作为除数。许多语言都没有这种特性，知道 number 为 0 后，仍继续检查后面的条件。

最后，考虑这个例子：

```
while ( x++ < 10 && x + y < 20)
```

实际上，&&是一个序列点，这保证了在对&&右侧的表达式求值之前，已经递增了 x。

小结：逻辑运算符和表达式

逻辑运算符：

逻辑运算符的运算对象通常是关系表达式。!运算符只需要一个运算对象，其他两个逻辑运算符都需要两个运算对象，左侧一个，右侧一个。

逻辑运算符	含义
&&	与
||	或
!	非

逻辑表达式：

当且仅当 expression1 和 expression2 都为真，expression1 && expression2 才为真。如果 expression1 或 expression2 为真，expression1 || expression2 为真。如果 expression 为假，!expression 则为真，反之亦然。

求值顺序：

逻辑表达式的求值顺序是从左往右。一旦发现有使整个表达式为假的因素，立即停止求值。

示例：

```
6 > 2 && 3 == 3          真
!(6 > 2 && 3 == 3)       假
x != 0 && (20 / x) < 5   只有当 x 不等于 0 时，才会对第 2 个表达式求值
```

7.3.4 范围

&&运算符可用于测试范围。例如，要测试 score 是否在 90～100 的范围内，可以这样写：

```
if (range >= 90 && range <= 100)
    printf("Good show!\n");
```

千万不要模仿数学上的写法：

```
if (90 <= range <= 100)    // 千万不要这样写！
    printf("Good show!\n");
```

这样写的问题是代码有语义错误，而不是语法错误，所以编译器不会捕获这样的问题（虽然可能会给出警告）。由于<=运算符的求值顺序是从左往右，所以编译器把测试表达式解释为：

```
(90 <= range) <= 100
```

子表达式 90 <= range 的值要么是 1（为真），要么是 0（为假）。这两个值都小于 100，所以不管 range 的值是多少，整个表达式都恒为真。因此，在范围测试中要使用&&。

许多代码都用范围测试来确定一个字符是否是小写字母。例如，假设 ch 是 char 类型的变量：

```
if (ch >= 'a' && ch <= 'z')
    printf("That's a lowercase character.\n");
```

该方法仅对于像 ASCII 这样的字符编码有效，这些编码中相邻字母与相邻数字一一对应。但是，对于像 EBCDIC 这样的代码就没用了。相应的可移植方法是，用 ctype.h 系列中的 islower()函数（参见表 7.1）：

```
if (islower(ch))
    printf("That's a lowercase character.\n");
```

无论使用哪种特定的字符编码，islower()函数都能正常运行（不过，一些早期的编译器没有 ctype.h 系列）。

7.4　一个统计单词的程序

现在，我们可以编写一个统计单词数量的程序（即，该程序读取并报告单词的数量）。该程序还可以计算字符数和行数。先来看看编写这样的程序要涉及那些内容。

首先，该程序要逐个字符读取输入，知道何时停止读取。然后，该程序能识别并计算这些内容：字符、行数和单词。据此我们编写的伪代码如下：

读取一个字符

当有更多输入时

　　递增字符计数

　　如果读完一行，递增行数计数

　　如果读完一个单词，递增单词计数

　　读取下一个字符

前面有一个输入循环的模型：

```
while ((ch = getchar()) != STOP)
{
    ...
}
```

这里，STOP 表示能标识输入末尾的某个值。以前我们用过换行符和句点标记输入的末尾，但是对于一个通用的统计单词程序，它们都不合适。我们暂时选用一个文本中不常用的字符（如，|）作为输入的末尾标记。第 8 章中会介绍更好的方法，以便程序既能处理文本文件，又能处理键盘输入。

现在，我们考虑循环体。因为该程序使用 getchar()进行输入，所以每次迭代都要通过递增计数器来计数。为了统计行数，程序要能检查换行字符。如果输入的字符是一个换行符，该程序应该递增行数计数器。这里要注意 STOP 字符位于一行的中间的情况。是否递增行数计数？我们可以作为特殊行计数，即没有换行符的一行字符。可以通过记录之前读取的字符识别这种情况，即如果读取时发现 STOP 字符的上一个字符不是换行符，那么这行就是特殊行。

最棘手的部分是识别单词。首先，必须定义什么是该程序识别的单词。我们用一个相对简单的方法，把一个单词定义为一个不含空白（即，没有空格、制表符或换行符）的字符序列。因此，“glymxck”和“r2d2”都算是一个单词。程序读取的第 1 个非空白字符即是一个单词的开始，当读到空白字符时结束。判断非空白字符最直接的测试表达式是：

```
c != ' ' && c != '\n' && c != '\t' /* 如果 c 不是空白字符，该表达式为真*/
```

检测空白字符最直接的测试表达式是：

```
c == ' ' || c == '\n' || c == '\t' /*如果 c 是空白字符，该表达式为真*/
```

然而，使用 ctype.h 头文件中的函数 isspace()更简单，如果该函数的参数是空白字符，则返回真。所以，如果 c 是空白字符，isspace(c)为真；如果 c 不是空白字符，!isspace(c)为真。

要查找一个单词里是否有某个字符，可以在程序读入单词的首字符时把一个标记（名为 inword）设置为 1。也可以在此时递增单词计数。然后，只要 inword 为 1（或 true），后续的非空白字符都不记为单词的开始。下一个空白字符，必须重置标记为 0（或 false），然后程序就准备好读取下一个单词。我们

把以上分析写成伪代码：

如果 c 不是空白字符，且 inword 为假

　　设置 inword 为真，并给单词计数

如果 c 是空白字符，且 inword 为真

　　设置 inword 为假

这种方法在读到每个单词的开头时把 inword 设置为 1（真），在读到每个单词的末尾时把 inword 设置为 0（假）。只有在标记从 0 设置为 1 时，递增单词计数。如果能使用_Bool 类型，可以在程序中包含 stdbool.h 头文件，把 inword 的类型设置为 bool，其值用 true 和 false 表示。如果编译器不支持这种用法，就把 inword 的类型设置为 int，其值用 1 和 0 表示。

如果使用布尔类型的变量，通常习惯把变量自身作为测试条件。如下所示：

用 if (inword)代替 if (inword == true)

用 if (!inword)代替 if (inword == false)

可以这样做的原因是，如果 inword 为 true，则表达式 inword == true 为 true；如果 inword 为 false，则表达式 inword == true 为 false。所以，还不如直接用 inword 作为测试条件。类似地，!inword 的值与表达式 inword == false 的值相同（非真即 false，非假即 true）。

程序清单 7.7 把上述思路（识别行、识别不完整的行和识别单词）翻译了成 C 代码。

程序清单 7.7　wordcnt.c 程序

```
// wordcnt.c -- 统计字符数、单词数、行数
#include <stdio.h>
#include <ctype.h>          // 为 isspace()函数提供原型
#include <stdbool.h>        // 为 bool、true、false 提供定义
#define STOP '|'
int main(void)
{
    char c;                 // 读入字符
    char prev;              // 读入的前一个字符
    long n_chars = 0L;// 字符数
    int n_lines = 0;        // 行数
    int n_words = 0;        // 单词数
    int p_lines = 0;        // 不完整的行数
    bool inword = false;    // 如果 c 在单词中，inword 等于 true

    printf("Enter text to be analyzed (| to terminate):\n");
    prev = '\n';            // 用于识别完整的行
    while ((c = getchar()) != STOP)
    {
        n_chars++;          // 统计字符
        if (c == '\n')
            n_lines++;      // 统计行
        if (!isspace(c) && !inword)
        {
            inword = true;// 开始一个新的单词
            n_words++;      // 统计单词
        }
```

```
            if (isspace(c) && inword)
                inword = false;    // 打到单词的末尾
            prev = c;          // 保存字符的值
        }

        if (prev != '\n')
            p_lines = 1;
        printf("characters = %ld, words = %d, lines = %d, ",
                n_chars, n_words, n_lines);
        printf("partial lines = %d\n", p_lines);

        return 0;
    }
```

下面是运行该程序后的一个输出示例：

```
Enter text to be analyzed (| to terminate):
Reason is a
powerful servant but
an inadequate master.
|
characters = 55, words = 9, lines = 3, partial lines = 0
```

该程序使用逻辑运算符把伪代码翻译成 C 代码。例如，把下面的伪代码：

如果 c 不是空白字符，且 inword 为假

翻译成如下 C 代码：

```
if (!isspace(c) &&!inword)
```

再次提醒读者注意，`!inword` 与 `inword == false` 等价。上面的整个测试条件比单独判断每个空白字符的可读性高：

```
if (c != ' ' && c != '\n' && c != '\t' && !inword)
```

上面的两种形式都表示“如果 c **不是**空白字符，且如果 c **不在**单词里”。如果两个条件都满足，则一定是一个新单词的开头，所以要递增 `n_words`。如果位于单词中，满足第 1 个条件，但是 `inword` 为 `true`，就不递增 `n_word`。当读到下一个空白字符时，`inword` 被再次设置为 `false`。检查代码，查看一下如果单词之间有多个空格时，程序是否能正常运行。第 8 章讲解了如何修正这个问题，让该程序能统计文件中的单词量。

7.5　条件运算符：?:

C 提供**条件表达式**（*conditional expression*）作为表达 `if else` 语句的一种便捷方式，该表达式使用?:条件运算符。该运算符分为两部分，需要 3 个运算对象。回忆一下，带一个运算对象的运算符称为一元运算符，带两个运算对象的运算符称为二元运算符。以此类推，带 3 个运算对象的运算符称为三元运算符。条件运算符是 C 语言中唯一的三元运算符。下面的代码得到一个数的绝对值：

```
x = (y < 0) ? -y : y;
```

在=和;之间的内容就是条件表达式，该语句的意思是“如果 y 小于 0，那么 `x = -y;`否则，`x = y`”。用 `if else` 可以这样表达：

```
if (y < 0)
    x = -y;
else
    x = y;
```

条件表达式的通用形式如下：

expression1 ? expression2 : expression3

如果 *expression1* 为真（非 0），那么整个条件表达式的值与 *expression2* 的值相同；如果 *expression1* 为假（0），那么整个条件表达式的值与 *expression3* 的值相同。

需要把两个值中的一个赋给变量时，就可以用条件表达式。典型的例子是，把两个值中的最大值赋给变量：

```
max = (a > b) ? a : b;
```

如果 a 大于 b，那么将 max 设置为 a；否则，设置为 b。

通常，条件运算符完成的任务用 if else 语句也可以完成。但是，使用条件运算符的代码更简洁，而且编译器可以生成更紧凑的程序代码。

我们来看程序清单 7.8 中的油漆程序，该程序计算刷给定平方英尺的面积需要多少罐油漆。基本算法很简单：用平方英尺数除以每罐油漆能刷的面积。但是，商店只卖整罐油漆，不会拆分来卖，所以如果计算结果是 1.7 罐，就需要两罐。因此，该程序计算得到带小数的结果时应该进 1。条件运算符常用于处理这种情况，而且还要根据单复数分别打印 can 和 cans。

程序清单 7.8　paint.c 程序

```
/* paint.c -- 使用条件运算符 */
#include <stdio.h>
#define COVERAGE 350      // 每罐油漆可刷的面积（单位：平方英尺）
int main(void)
{
    int sq_feet;
    int cans;

    printf("Enter number of square feet to be painted:\n");
    while (scanf("%d", &sq_feet) == 1)
    {
        cans = sq_feet / COVERAGE;
        cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
        printf("You need %d %s of paint.\n", cans,
               cans == 1 ? "can" : "cans");
        printf("Enter next value (q to quit):\n");
    }

    return 0;
}
```

下面是该程序的运行示例：

```
Enter number of square feet to be painted:
349
You need 1 can of paint.
Enter next value (q to quit):
351
You need 2 cans of paint.
Enter next value (q to quit):
q
```

该程序使用的变量都是 int 类型，除法的计算结果（sq_feet / COVERAGE）会被截断。也就是说，351/350 得 1。所以，cans 被截断成整数部分。如果 sq_feet % COVERAGE 得 0，说明 sq_feet 被

COVERAGE 整除，cans 的值不变；否则，肯定有余数，就要给 cans 加 1。这由下面的语句完成：

```
cans += ((sq_feet % COVERAGE == 0)) ? 0 : 1;
```

该语句把+=右侧表达式的值加上 cans，再赋给 cans。右侧表达式是一个条件表达式，根据 sq_feet 是否能被 COVERAGE 整除，其值为 0 或 1。

printf()函数中的参数也是一个条件表达式：

```
cans == 1 ? "can" : "cans");
```

如果 cans 的值是 1，则打印 can；否则，打印 cans。这也说明了条件运算符的第 2 个和第 3 个运算对象可以是字符串。

小结：条件运算符

条件运算符：?:

一般注解：

条件运算符需要 3 个运算对象，每个运算对象都是一个表达式。其通用形式如下：

expression1 ? expression2 : expression3

如果 *expression1* 为真，整个条件表达式的值是 *expression2* 的值；否则，是 *expression3* 的值。

示例：

(5 > 3) ? 1 : 2 值为 1

(3 > 5) ? 1 : 2 值为 2

(a > b) ? a : b 如果 a >b，则取较大的值

7.6　循环辅助：continue 和 break

一般而言，程序进入循环后，在下一次循环测试之前会执行完循环体中的所有语句。continue 和 break 语句可以根据循环体中的测试结果来忽略一部分循环内容，甚至结束循环。

7.6.1　continue 语句

3 种循环都可以使用 continue 语句。执行到该语句时，会跳过本次迭代的剩余部分，并开始下一轮迭代。如果 continue 语句在嵌套循环内，则只会影响包含该语句的内层循环。程序清单 7.9 中的简短程序演示了如何使用 continue。

程序清单 7.9　skippart.c 程序

```
/* skippart.c  -- 使用 continue 跳过部分循环 */
#include <stdio.h>
int main(void)
{
    const float MIN = 0.0f;
    const float MAX = 100.0f;

    float score;
    float total = 0.0f;
    int n = 0;
    float min = MAX;
```

```
    float max = MIN;

    printf("Enter the first score (q to quit): ");
    while (scanf("%f", &score) == 1)
    {
        if (score < MIN || score > MAX)
        {
            printf("%0.1f is an invalid value. Try again: ",score);
            continue;   // 跳转至 while 循环的测试条件
        }
        printf("Accepting %0.1f:\n", score);
        min = (score < min) ? score : min;
        max = (score > max) ? score : max;
        total += score;
        n++;
        printf("Enter next score (q to quit): ");
    }
    if (n > 0)
    {
        printf("Average of %d scores is %0.1f.\n", n, total / n);
        printf("Low = %0.1f, high = %0.1f\n", min, max);
    }
    else
        printf("No valid scores were entered.\n");
    return 0;
}
```

在程序清单 7.9 中，while 循环读取输入，直至用户输入非数值数据。循环中的 if 语句筛选出无效的分数。假设输入 188，程序会报告：188 is an invalid value。在本例中，continue 语句让程序跳过处理有效输入部分的代码。程序开始下一轮循环，准备读取下一个输入值。

注意，有两种方法可以避免使用 continue，一是省略 continue，把剩余部分放在一个 else 块中：

```
if (score < 0 || score > 100)
    /* printf()语句 */
else
{
    /* 语句*/
}
```

另一种方法是，用以下格式来代替：

```
if (score >= 0 && score <= 100)
{
    /* 语句 */
}
```

这种情况下，使用 continue 的好处是减少主语句组中的一级缩进。当语句很长或嵌套较多时，紧凑简洁的格式提高了代码的可读性。

continue 还可用作占位符。例如，下面的循环读取并丢弃输入的数据，直至读到行末尾：

```
while (getchar() != '\n')
    ;
```

当程序已经读取一行中的某些内容，要跳至下一行开始处时，这种用法很方便。问题是，一般很难注意到一个单独的分号。如果使用 continue，可读性会更高：

```
while (getchar() != '\n')
    continue;
```

如果用了 continue 没有简化代码反而让代码更复杂，就不要使用 continue。例如，考虑下面的程序段：

```
while ((ch = getchar() ) != '\n')
{
    if (ch == '\t')
        continue;
    putchar(ch);
}
```

该循环跳过制表符，并在读到换行符时退出循环。以上代码这样表示更简洁：

```
while ((ch = getchar()) != '\n')
    if (ch != '\t')
        putchar(ch);
```

通常，在这种情况下，把 if 的测试条件的关系反过来便可避免使用 continue。

以上介绍了 continue 语句让程序跳过循环体的余下部分。那么，从何处开始继续循环？对于 while 和 do while 循环，执行 continue 语句后的下一个行为是对循环的测试表达式求值。考虑下面的循环：

```
count = 0;
while (count < 10)
{
    ch = getchar();
    if (ch == '\n')
        continue;
    putchar(ch);
    count++;
}
```

该循环读取 10 个字符（除换行符外，因为当 ch 是换行符时，程序会跳过 count++;语句）并重新显示它们，其中不包括换行符。执行 continue 后，下一个被求值的表达式是循环测试条件。

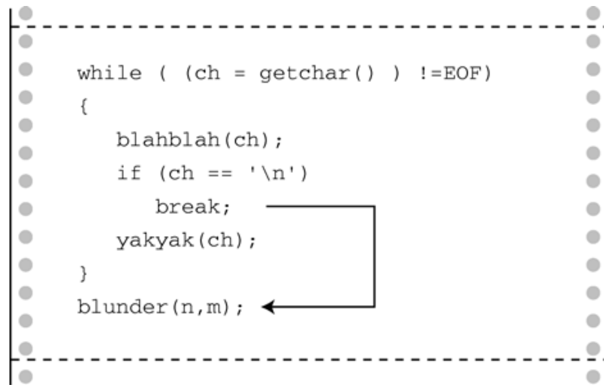
对于 for 循环，执行 continue 后的下一个行为是对更新表达式求值，然后是对循环测试表达式求值。例如，考虑下面的循环：

```
for (count = 0; count < 10; count++)
{
    ch = getchar();
    if (ch == '\n')
        continue;
    putchar(ch);
}
```

该例中，执行完 continue 后，首先递增 count，然后将递增后的值和 10 作比较。因此，该循环与上面 while 循环的例子稍有不同。while 循环的例子中，除了换行符，其余字符都显示；而本例中，换行符也计算在内，所以读取的 10 个字符中包含换行符。

7.6.2　break 语句

程序执行到循环中的 break 语句时，会终止包含它的循环，并继续执行下一阶段。把程序清单 7.9 中的 continue 替换成 break，在输入 188 时，不是跳至执行下一轮循环，而是导致退出当前循环。图 7.3 比较了 break 和 continue。如果 break 语句位于嵌套循环内，它只会影响包含它的当前循环。



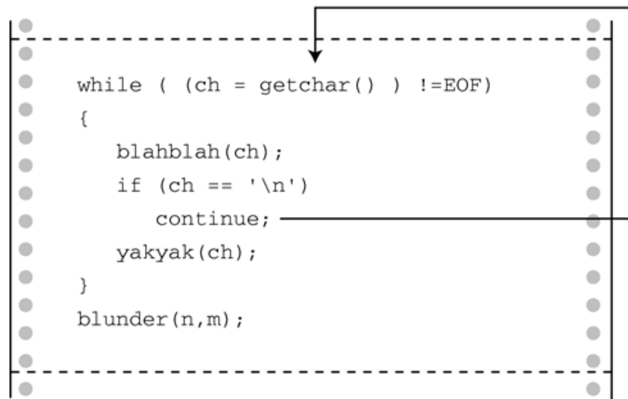


图 7.3　比较 break 和 continue

break 还可用于因其他原因退出循环的情况。程序清单 7.10 用一个循环计算矩形的面积。如果用户输入非数字作为矩形的长或宽，则终止循环。

程序清单 7.10　break.c 程序

```
/* break.c -- 使用 break 退出循环 */
#include <stdio.h>
int main(void)
{
    float length, width;

    printf("Enter the length of the rectangle:\n");
    while (scanf("%f", &length) == 1)
    {
        printf("Length = %0.2f:\n", length);
        printf("Enter its width:\n");
        if (scanf("%f", &width) != 1)
            break;
        printf("Width = %0.2f:\n", width);
        printf("Area = %0.2f:\n", length * width);
        printf("Enter the length of the rectangle:\n");
    }
    printf("Done.\n");
```

```
    return 0;
}
```

可以这样控制循环：

```
while (scanf("%f %f", &length, &width) == 2)
```

但是，用 break 可以方便显示用户输入的值。

和 continue 一样，如果用了 break 代码反而更复杂，就不要使用 break。例如，考虑下面的循环：

```
while ((ch = getchar()) != '\n')
{
    if (ch == '\t')
        break;
    putchar(ch);
}
```

如果把两个测试条件放在一起，逻辑就更清晰了：

```
while ((ch = getchar() ) != '\n' && ch != '\t')
    putchar(ch);
```

break 语句对于稍后讨论的 switch 语句而言至关重要。

在 for 循环中的 break 和 continue 的情况不同，执行完 break 语句后会直接执行循环后面的第 1 条语句，连更新部分也跳过。嵌套循环内层的 break 只会让程序跳出包含它的当前循环，要跳出外层循环还需要一个 break：

```
int p, q;

scanf("%d", &p);
while (p > 0)
{
    printf("%d\n", p);
    scanf("%d", &q);
    while (q > 0)
    {
        printf("%d\n", p*q);
        if (q > 100)
            break; // 跳出内层循环
        scanf("%d", &q);
    }
    if (q > 100)
        break; // 跳出外层循环
    scanf("%d", &p);
}
```

7.7　多重选择：switch 和 break

使用条件运算符和 if else 语句很容易编写二选一的程序。然而，有时程序需要在多个选项中进行选择。可以用 if else if...else 来完成。但是，大多数情况下使用 switch 语句更方便。程序清单 7.11 演示了如何使用 switch 语句。该程序读入一个字母，然后打印出与该字母开头的动物名。

程序清单 7.11　animals.c 程序

```
/* animals.c -- 使用 switch 语句 */
#include <stdio.h>
```

```
#include <ctype.h>
 int main(void)
{
    char ch;

    printf("Give me a letter of the alphabet, and I will give ");
    printf("an animal name\nbeginning with that letter.\n");
    printf("Please type in a letter; type # to end my act.\n");
    while ((ch = getchar()) != '#')
    {
        if ('\n' == ch)
            continue;
        if (islower(ch))    /* 只接受小写字母*/
            switch (ch)
        {
            case 'a':
                  printf("argali, a wild sheep of Asia\n");
                  break;
            case 'b':
                  printf("babirusa, a wild pig of Malay\n");
                  break;
            case 'c':
                  printf("coati, racoonlike mammal\n");
                  break;
            case 'd':
                  printf("desman, aquatic, molelike critter\n");
                  break;
            case 'e':
                  printf("echidna, the spiny anteater\n");
                  break;
            case 'f':
                  printf("fisher, brownish marten\n");
                  break;
            default:
                  printf("That's a stumper!\n");
        }                  /* switch 结束          */
        else
            printf("I recognize only lowercase letters.\n");
        while (getchar() != '\n')
            continue;      /* 跳过输入行的剩余部分   */
        printf("Please type another letter or a #.\n");
    }                      /* while 循环结束        */
    printf("Bye!\n");

    return 0;
}
```

篇幅有限，我们只编到 f，后面的字母以此类推。在进一步解释该程序之前，先看看输出示例：

```
Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
Please type in a letter; type # to end my act.
a [enter]
argali, a wild sheep of Asia
Please type another letter or a #.
```

```
dab [enter]
desman, aquatic, molelike critter
Please type another letter or a #.
r [enter]
That's a stumper!
Please type another letter or a #.
Q [enter]
I recognize only lowercase letters.
Please type another letter or a #.
# [enter]
Bye!
```

该程序的两个主要特点是：使用了 switch 语句和它对输出的处理。我们先分析 switch 的工作原理。

7.7.1　switch 语句

要对紧跟在关键字 switch 后圆括号中的表达式求值。在程序清单 7.11 中，该表达式是刚输入给 ch 的值。然后程序扫描**标签**（这里指，case 'a' :、case 'b' :等）列表，直到发现一个匹配的值为止。然后程序跳转至那一行。如果没有匹配的标签怎么办？如果有 default :标签行，就跳转至该行；否则，程序继续执行在 switch 后面的语句。

break 语句在其中起什么作用？它让程序离开 switch 语句，跳至 switch 语句后面的下一条语句（见图 7.4）。如果没有 break 语句，就会从匹配标签开始执行到 switch 末尾。例如，如果删除该程序中的所有 break 语句，运行程序后输入 d，其交互的输出结果如下：

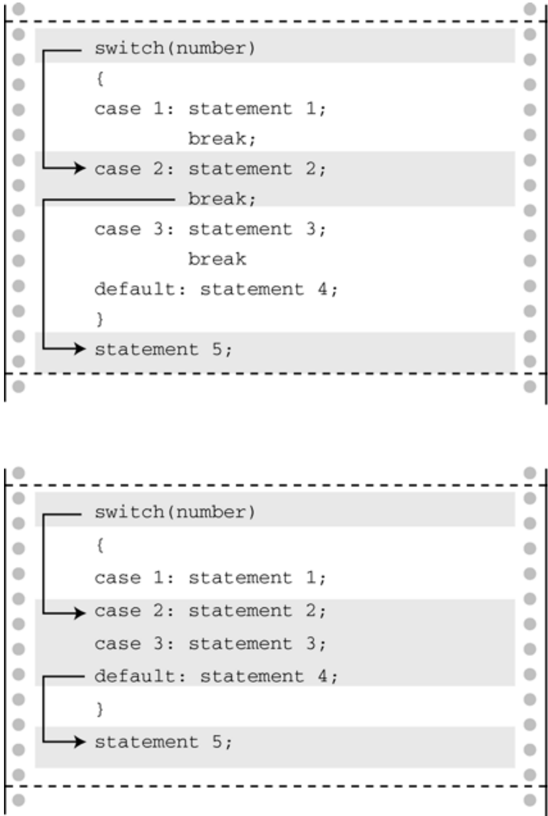


图 7.4　switch 中有 break 和没有 break 的程序流

```
Give me a letter of the alphabet, and I will give an animal name
beginning with that letter.
 Please type in a letter; type # to end my act.
d [enter]
desman, aquatic, molelike critter
echidna, the spiny anteater
fisher, a brownish marten
That's a stumper!
Please type another letter or a #.
# [enter]
Bye!
```

如上所示，执行了从 case 'd':到 switch 语句末尾的所有语句。

顺带一提，break 语句可用于循环和 switch 语句中，但是 continue 只能用于循环中。尽管如此，如果 switch 语句在一个循环中，continue 便可作为 switch 语句的一部分。这种情况下，就像在其他循环中一样，continue 让程序跳出循环的剩余部分，包括 switch 语句的其他部分。

如果读者熟悉 Pascal，会发现 switch 语句和 Pascal 的 case 语句类似。它们最大的区别在于，如果只希望处理某个带标签的语句，就必须在 switch 语句中使用 break 语句。另外，C 语言的 case 一般都指定一个值，不能使用一个范围。

switch 在圆括号中的测试表达式的值应该是一个整数值（包括 char 类型）。case 标签必须是整数类型（包括 char 类型）的常量或整型常量表达式（即，表达式中只包含整型常量）。不能用变量作为 case 标签。switch 的构造如下：

```
switch ( 整型表达式)
{
    case 常量1:
         语句      <--可选
    case 常量2:
         语句      <--可选
    default :     <--可选
         语句      <--可选
}
```

7.7.2 只读每行的首字符

animals.c（程序清单 7.11）的另一个独特之处是它读取输入的方式。运行程序时读者可能注意到了，当输入 dab 时，只处理了第 1 个字符。这种丢弃一行中其他字符的行为，经常出现在响应单字符的交互程序中。可以用下面的代码实现这样的行为：

```
while (getchar() != '\n')
    continue;          /* 跳过输入行的其余部分 */
```

循环从输入中读取字符，包括按下 Enter 键产生的换行符。注意，函数的返回值并没有赋给 ch，以上代码所做的只是读取并丢弃字符。由于最后丢弃的字符是换行符，所以下一个被读取的字符是下一行的首字母。在外层的 while 循环中，getchar()读取首字母并赋给 ch。

假设用户一开始就按下 Enter 键，那么程序读到的首个字符就是换行符。下面的代码处理这种情况：

```
if (ch == '\n')
    continue;
```

7.7.3　多重标签

如程序清单 7.12 所示，可以在 switch 语句中使用多重 case 标签。

程序清单 7.12　vowels.c 程序

```
// vowels.c -- 使用多重标签
#include <stdio.h>
int main(void)
{
    char ch;
    int a_ct, e_ct, i_ct, o_ct, u_ct;

    a_ct = e_ct = i_ct = o_ct = u_ct = 0;

    printf("Enter some text; enter # to quit.\n");
    while ((ch = getchar()) != '#')
    {
        switch (ch)
        {
            case 'a':
            case 'A':  a_ct++;
                       break;
            case 'e':
            case 'E':  e_ct++;
                       break;
            case 'i':
            case 'I':  i_ct++;
                       break;
            case 'o':
            case 'O':  o_ct++;
                       break;
            case 'u':
            case 'U':  u_ct++;
                       break;
            default:   break;
        }                  // switch 结束
    }                      // while 循环结束
    printf("number of vowels:   A    E    I    O    U\n");
    printf("              %4d %4d %4d %4d %4d\n",
            a_ct, e_ct, i_ct, o_ct, u_ct);

    return 0;
}
```

假设如果 ch 是字母 i，switch 语句会定位到标签为 case 'i' :的位置。由于该标签没有关联 break 语句，所以程序流直接执行下一条语句，即 i_ct++;。如果 ch 是字母 I，程序流会直接定位到 case 'I' :。本质上，两个标签都指的是相同的语句。

严格地说，case 'U'的 break 语句并不需要。因为即使删除这条 break 语句，程序流会接着执行 switch 中的下一条语句，即 default : break;。所以，可以把 case 'U'的 break 语句去掉以缩短代码。但是从另一方面看，保留这条 break 语句可以防止以后在添加新的 case（例如，把 y 作为元音）时遗漏 break 语句。

下面是该程序的运行示例：

```
Enter some text; enter # to quit.
I see under the overseer.#
number of vowels:     A    E    I    O    U
                      0    7    1    1    1
```

在该例中，如果使用 ctype.h 系列的 toupper()函数（参见表 7.2）可以避免使用多重标签，在进行测试之前就把字母转换成大写字母：

```
while ((ch = getchar()) != '#')
{
    ch = toupper(ch);
    switch (ch)
    {
        case 'A': a_ct++;
                  break;
        case 'E': e_ct++;
                  break;
        case 'I': i_ct++;
                  break;
        case 'O': o_ct++;
                  break;
        case 'U': u_ct++;
                  break;
        default: break;
    } // switch 结束
} // while 循环结束
```

或者，也可以先不转换 ch，把 toupper(ch)放进 switch 的测试条件中：switch(toupper(ch))。

小结：带多重选择的 switch 语句

关键字： switch

一般注解：

程序根据 *expression* 的值跳转至相应的 case 标签处。然后，执行剩下的所有语句，除非执行到 break 语句进行重定向。*expression* 和 case 标签都必须是整数值（包括 char 类型），标签必须是常量或完全由常量组成的表达式。如果没有 case 标签与 *expression* 的值匹配，控制则转至标有 default 的语句（如果有的话）；否则，将转至执行紧跟在 switch 语句后面的语句。

形式：

```
switch ( expression )
{
    case label1 : statement1//使用 break 跳出 switch
    case label2 : statement2
    default     : statement3
}
```

可以有多个标签语句，default 语句可选。

示例：

```
switch (choice)
    {
    case 1 :
    case 2 : printf("Darn tootin'!\n"); break;
    case 3 : printf("Quite right!\n");
    case 4 : printf("Good show!\n"); break;
```

```
    default: printf("Have a nice day.\n");
    }
```

如果 choice 的值是 1 或 2，打印第 1 条消息；如果 choice 的值是 3，打印第 2 条和第 3 条消息（程序继续执行后续的语句，因为 case 3 后面没有 break 语句）；如果 choice 的值是 4，则打印第 3 条消息；如果 choice 的值是其他值只打印最后一条消息。

7.7.4　switch 和 if else

何时使用 switch？何时使用 if else？你经常会别无选择。如果是根据浮点类型的变量或表达式来选择，就无法使用 switch。如果根据变量在某范围内决定程序流的去向，使用 switch 就很麻烦，这种情况用 if 就很方便：

```
if (integer < 1000 && integer > 2)
```

使用 switch 要涵盖以上范围，需要为每个整数（3～999）设置 case 标签。但是，如果使用 switch，程序通常运行快一些，生成的代码少一些。

7.8　goto 语句

早期版本的 BASIC 和 FORTRAN 所依赖的 goto 语句，在 C 中仍然可用。但是 C 和其他两种语言不同，没有 goto 语句 C 程序也能运行良好。Kernighan 和 Ritchie 提到 goto 语句“易被滥用”，并建议“谨慎使用，或者根本不用”。首先，介绍一下如何使用 goto 语句；然后，讲解为什么通常不需要它。

goto 语句有两部分：goto 和标签名。标签的命名遵循变量命名规则，如下所示：

```
goto part2;
```

要让这条语句正常工作，函数还必须包含另一条标为 part2 的语句，该语句以标签名后紧跟一个冒号开始：

```
part2: printf("Refined analysis:\n");
```

7.8.1　避免使用 goto

原则上，根本不用在 C 程序中使用 goto 语句。但是，如果你曾经学过 FORTRAN 或 BASIC（goto 对这两种语言而言都必不可少），可能还会依赖用 goto 来编程。为了帮助你克服这个习惯，我们先概述一些使用 goto 的常见情况，然后再介绍 C 的解决方案。

■ 处理包含多条语句的 if 语句：

```
if (size > 12)
    goto a;
goto b;
a: cost = cost * 1.05;
flag = 2;
b: bill = cost * flag;
```

对于以前的 BASIC 和 FORTRAN，只有直接跟在 if 条件后面的一条语句才属于 if，不能使用块或复合语句。我们把以上模式转换成等价的 C 代码，标准 C 用复合语句或块来处理这种情况：

```
if (size > 12)
{
    cost = cost * 1.05;
    flag = 2;
```

```
}
bill = cost * flag;
```

- 二选一：

```
if (ibex > 14)
    goto a;
sheds = 2;
goto b;
a: sheds= 3;
b: help = 2 * sheds;
```

C 通过 if else 表达二选一更清楚：

```
if (ibex > 14)
    sheds = 3;
else
    sheds = 2;
help = 2 * sheds;
```

实际上，新版的 BASIC 和 FORTRAN 已经把 else 纳入新的语法中。

- 创建不确定循环：

```
readin: scanf("%d", &score);
if (score < 0)
    goto stage2;
lots of statements
goto readin;
stage2: more stuff;
```

C 用 while 循环代替：

```
scanf("%d", &score);
while (score <= 0)
{
   lots of statements
   scanf("%d", &score);
}
more stuff;
```

- 跳转至循环末尾，并开始下一轮迭代。C 使用 continue 语句代替。
- 跳出循环。C 使用 break 语句。实际上，break 和 continue 是 goto 的特殊形式。使用 break 和 continue 的好处是：其名称已经表明它们的用法，而且这些语句不使用标签，所以不用担心把标签放错位置导致的危险。
- 胡乱跳转至程序的不同部分。简而言之，不要这样做！

但是，C 程序员可以接受一种 goto 的用法——出现问题时从一组嵌套循环中跳出（一条 break 语句只能跳出当前循环）：

```
while (funct > 0)
{
    for (i = 1, i <= 100; i++)
        {
        for (j = 1; j <= 50; j++)
        {
            其他语句
            if (问题)
             goto help;
            其他语句
```

```
            }
            其他语句
        }
        其他语句
    }
    其他语句
    help: 语句
```

从其他例子中也能看出，程序中使用其他形式比使用 goto 的条理更清晰。当多种情况混在一起时，这种差异更加明显。哪些 goto 语句可以帮助 if 语句？哪些可以模仿 if else？哪些控制循环？哪些是因为程序无路可走才不得已放在那里？过度地使用 goto 语句，会让程序错综复杂。如果不熟悉 goto 语句，就不要使用它。如果已经习惯使用 goto 语句，试着改掉这个毛病。讽刺地是，虽然 C 根本不需要 goto，但是它的 goto 比其他语言的 goto 好用，因为 C 允许在标签中使用描述性的单词而不是数字。

小结：程序跳转

关键字：break、continue、goto

一般注解：

这 3 种语句都能使程序流从程序的一处跳转至另一处。

break 语句：

所有的循环和 switch 语句都可以使用 break 语句。它使程序控制跳出当前循环或 switch 语句的剩余部分，并继续执行跟在循环或 switch 后面的语句。

示例：

```
switch (number)
{
    case 4: printf("That's a good choice.\n");
            break;
    case 5: printf("That's a fair choice.\n");
            break;
    default: printf("That's a poor choice.\n");
}
```

continue 语句：

所有的循环都可以使用 continue 语句，但是 switch 语句不行。continue 语句使程序控制跳出循环的剩余部分。对于 while 或 for 循环，程序执行到 continue 语句后会开始进入下一轮迭代。对于 do while 循环，对出口条件求值后，如有必要会进入下一轮迭代。

示例：

```
while ((ch = getchar()) != '\n')
{
    if (ch == ' ')
        continue;
    putchar(ch);
    chcount++;
}
```

以上程序段把用户输入的字符再次显示在屏幕上，并统计非空格字符。

goto 语句：

goto 语句使程序控制跳转至相应标签语句。冒号用于分隔标签和标签语句。标签名遵循变量命名规则。标签语句可以出现在 goto 的前面或后面。

```
形式：
goto label ;
    .
    .
    .
label : statement
示例：
top : ch = getchar();
    .
    .
    .
if (ch != 'y')
goto top;
```

7.9　关键概念

智能的一个方面是，根据情况做出相应的响应。所以，选择语句是开发具有智能行为程序的基础。C 语言通过 if、if else 和 switch 语句，以及条件运算符（?:）可以实现智能选择。

if 和 if else 语句使用测试条件来判断执行哪些语句。所有非零值都被视为 true，零被视为 false。测试通常涉及关系表达式（比较两个值）、逻辑表达式（用逻辑运算符组合或更改其他表达式）。

要记住一个通用原则，如果要测试两个条件，应该使用逻辑运算符把两个完整的测试表达式组合起来。例如，下面这些是错误的：

```
if (a < x < z)              // 错误，没有使用逻辑运算符
...
if (ch != 'q' && != 'Q')    // 错误，缺少完整的测试表达式
...
```

正确的方式是用逻辑运算符连接两个关系表达式：

```
if (a < x && x < z)            // 使用&&组合两个表达式
...
if (ch != 'q' && ch != 'Q')    // 使用&&组合两个表达式
...
```

对比这两章和前几章的程序示例可以发现：使用第 6 章、第 7 章介绍的语句，可以写出功能更强大、更有趣的程序。

7.10　本章小结

本章介绍了很多内容，我们来总结一下。if 语句使用测试条件控制程序是否执行测试条件后面的一条简单语句或复合语句。如果测试表达式的值是非零值，则执行语句；如果测试表达式的值是零，则不执行语句。if else 语句可用于二选一的情况。如果测试条件是非零，则执行 else 前面的语句；如果测试表达式的值是零，则执行 else 后面的语句。在 else 后面使用另一个 if 语句形成 else if，可构造多选一的结构。

测试条件通常都是**关系表达式**，即用一个关系运算符（如，<或==）的表达式。使用 C 的逻辑运算符，可以把关系表达式组合成更复杂的测试条件。

在多数情况下，用**条件运算符**（?:）写成的表达式比 if else 语句更简洁。

ctype.h 系列的字符函数（如，issapce()和 isalpha()）为创建以分类字符为基础的测试表达式提供了便捷的工具。

switch 语句可以在一系列以整数作为标签的语句中进行选择。如果紧跟在 switch 关键字后的测试条件的整数值与某标签匹配，程序就转至执行匹配的标签语句，然后在遇到 break 之前，继续执行标签语句后面的语句。

break、continue 和 goto 语句都是跳转语句，使程序流跳转至程序的另一处。break 语句使程序跳转至紧跟在包含 break 语句的循环或 switch 末尾的下一条语句。continue 语句使程序跳出当前循环的剩余部分，并开始下一轮迭代。

7.11　复习题

复习题的参考答案在附录 A 中。

1. 判断下列表达式是 true 还是 false。

   a. 100 > 3 && 'a'>'c'
   b. 100 > 3 || 'a'>'c'
   c. !(100>3)

2. 根据下列描述的条件，分别构造一个表达式：

   a. number等于或大于90，但是小于100
   b. ch不是字符q或k
   c. number在1～9之间（包括1和9），但不是5
   d. number不在1～9之间

3. 下面的程序关系表达式过于复杂，而且还有些错误，请简化并改正。

```
#include <stdio.h>
int main(void)                                          /* 1 */
{                                                       /* 2 */
    int weight, height; /* weight 以磅为单位, height 以英寸为单位 */
/* 4 */
    scanf("%d , weight, height);                        /* 5 */
    if (weight < 100 && height > 64)                    /* 6 */
        if (height >= 72)                               /* 7 */
            printf("You are very tall for your weight.\n");
        else if (height < 72 &&> 64)                    /* 9 */
            printf("You are tall for your weight.\n");/* 10 */
        else if (weight > 300 && !(weight <= 300)       /* 11 */
            && height < 48)                             /* 12 */
        if (!(height >= 48))                            /* 13 */
            printf(" You are quite short for your weight.\n");
    else                                                /* 15 */
        printf("Your weight is ideal.\n");              /* 16 */
                                                        /* 17 */
    return 0;
}
```

4. 下列个表达式的值是多少？

   a. 5 > 2
   b. 3 + 4 > 2 && 3 < 2
   c. x >= y || y > x

d. `d = 5 + ( 6 > 2 )`

e. `'X' > 'T' ? 10 : 5`

f. `x > y ? y > x : x > y`

5. 下面的程序将打印什么？

```
#include <stdio.h>
int main(void)
{
    int num;
    for (num = 1; num <= 11; num++)
    {
        if (num % 3 == 0)
            putchar('$');
        else
            putchar('*');
            putchar('#');
        putchar('%');
    }
    putchar('\n');
    return 0;
}
```

6. 下面的程序将打印什么？

```
#include <stdio.h>
int main(void)
{
    int i = 0;
    while (i < 3) {
        switch (i++) {
            case 0: printf("fat ");
            case 1: printf("hat ");
            case 2: printf("cat ");
            default: printf("Oh no!");
        }
        putchar('\n');
    }
    return 0;
}
```

7. 下面的程序有哪些错误？

```
#include <stdio.h>
int main(void)
{
    char ch;
    int lc = 0; /* 统计小写字母
    int uc = 0; /* 统计大写字母
    int oc = 0; /* 统计其他字母

    while ((ch = getchar()) != '#')
    {
        if ('a' <= ch >= 'z')
            lc++;
        else if (!(ch < 'A') || !(ch > 'Z')
            uc++;
        oc++;
```

```
    }
    printf(%d lowercase, %d uppercase, %d other, lc, uc, oc);
    return 0;
}
```

8. 下面的程序将打印什么？

```
/* retire.c */
#include <stdio.h>
int main(void)
{
    int age = 20;
    while (age++ <= 65)
    {
        if ((age % 20) == 0) /* age 是否能被 20 整除？ */
            printf("You are %d. Here is a raise.\n", age);
        if (age = 65)
            printf("You are %d. Here is your gold watch.\n", age);
    }
    return 0;
}
```

9. 给定下面的输入时，以下程序将打印什么？

```
q
c
h
b
#include <stdio.h>
int main(void)
{
    char ch;

    while ((ch = getchar()) != '#')
    {
        if (ch == '\n')
            continue;
        printf("Step 1\n");
        if (ch == 'c')
            continue;
        else if (ch == 'b')
            break;
        else if (ch == 'h')
            goto laststep;
        printf("Step 2\n");
    laststep: printf("Step 3\n");
    }
    printf("Done\n");
    return 0;
}
```

10. 重写复习题 9，但这次不能使用 continue 和 goto 语句。

7.12　编程练习

1. 编写一个程序读取输入，读到#字符停止，然后报告读取的空格数、换行符数和所有其他字符的数量。

2. 编写一个程序读取输入，读到#字符停止。程序要打印每个输入的字符以及对应的 ASCII 码（十进制）。一行打印 8 个字符。建议:使用字符计数和求模运算符（%）在每 8 个循环周期时打印一个换行符。

3. 编写一个程序，读取整数直到用户输入 0。输入结束后，程序应报告用户输入的偶数（不包括 0）个数、这些偶数的平均值、输入的奇数个数及其奇数的平均值。

4. 使用 if else 语句编写一个程序读取输入，读到#停止。用感叹号替换句号，用两个感叹号替换原来的感叹号，最后报告进行了多少次替换。

5. 使用 switch 重写练习 4。

6. 编写程序读取输入，读到#停止，报告 ei 出现的次数。

注意

该程序要记录前一个字符和当前字符。用“Receive your eieio award”这样的输入来测试。

7. 编写一个程序，提示用户输入一周工作的小时数，然后打印工资总额、税金和净收入。做如下假设：

   a．基本工资 = 1000 美元/小时

   b．加班（超过 40 小时） = 1.5 倍的时间

   c．税率： 前 300 美元为 15%

   续 150 美元为 20%

   余下的为 25%

   用#define 定义符号常量。不用在意是否符合当前的税法。

8. 修改练习 7 的假设 a，让程序可以给出一个供选择的工资等级菜单。使用 switch 完成工资等级选择。运行程序后，显示的菜单应该类似这样：

```
*****************************************************************
Enter the number corresponding to the desired pay rate or action:
1) $8.75/hr                           2) $9.33/hr
3) $10.00/hr                          4) $11.20/hr
5) quit
*****************************************************************
```

   如果选择 1～4 其中的一个数字，程序应该询问用户工作的小时数。程序要通过循环运行，除非用户输入 5。如果输入 1～5 以外的数字，程序应提醒用户输入正确的选项，然后再重复显示菜单提示用户输入。使用#define 创建符号常量表示各工资等级和税率。

9. 编写一个程序，只接受正整数输入，然后显示所有小于或等于该数的素数。

10. 1988 年的美国联邦税收计划是近代最简单的税收方案。它分为 4 个类别，每个类别有两个等级。下面是该税收计划的摘要（美元数为应征税的收入）：

类别	税金
单身	17850 美元按 15%计，超出部分按 28%计
户主	23900 美元按 15%计，超出部分按 28%计
已婚，共有	29750 美元按 15%计，超出部分按 28%计
已婚，离异	14875 美元按 15%计，超出部分按 28%计

例如，一位工资为 20000 美元的单身纳税人，应缴纳税费 0.15×17850+0.28×（20000−17850）美元。编写一个程序，让用户指定缴纳税金的种类和应纳税收入，然后计算税金。程序应通过循环让用户可以多次输入。

11. ABC 邮购杂货店出售的洋蓟售价为 2.05 美元/磅，甜菜售价为 1.15 美元/磅，胡萝卜售价为 1.09 美元/磅。在添加运费之前，100 美元的订单有 5%的打折优惠。少于或等于 5 磅的订单收取 6.5 美元的运费和包装费，5 磅～20 磅的订单收取 14 美元的运费和包装费，超过 20 磅的订单在 14 美元的基础上每续重 1 磅增加 0.5 美元。编写一个程序，在循环中用 `switch` 语句实现用户输入不同的字母时有不同的响应，即输入 `a` 的响应是让用户输入洋蓟的磅数，`b` 是甜菜的磅数，`c` 是胡萝卜的磅数，`q` 是退出订购。程序要记录累计的重量。即，如果用户输入 4 磅的甜菜，然后输入 5 磅的甜菜，程序应报告 9 磅的甜菜。然后，该程序要计算货物总价、折扣（如果有的话）、运费和包装费。随后，程序应显示所有的购买信息：物品售价、订购的重量（单位：磅）、订购的蔬菜费用、订单的总费用、折扣（如果有的话）、运费和包装费，以及所有的费用总额。

第 8 章

字符输入/输出和输入验证

本章介绍以下内容：

- 更详细地介绍输入、输出以及缓冲输入和无缓冲输入的区别
- 如何通过键盘模拟文件结尾条件
- 如何使用重定向把程序和文件相连接
- 创建更友好的用户界面

在涉及计算机的话题时，我们经常会提到输入（*input*）和输出（*output*）。我们谈论输入和输出设备（如键盘、U 盘、扫描仪和激光打印机），讲解如何处理输入数据和输出数据，讨论执行输入和输出任务的函数。本章主要介绍用于输入和输出的函数（简称 I/O 函数）。

I/O 函数（如`printf()`、`scanf()`、`getchar()`、`putchar()`等）负责把信息传送到程序中。前几章简单介绍过这些函数，本章将详细介绍它们的基本概念。同时，还会介绍如何设计与用户交互的界面。

最初，输入/输出函数不是 C 定义的一部分，C 把开发这些函数的任务留给编译器的实现者来完成。在实际应用中，UNIX 系统中的 C 实现为这些函数提供了一个模型。ANSI C 库吸取成功的经验，把大量的 UNIX I/O 函数囊括其中，包括一些我们曾经用过的。由于必须保证这些标准函数在不同的计算机环境中能正常工作，所以它们很少使用某些特殊系统才有的特性。因此，许多 C 供应商会利用硬件的特性，额外提供一些 I/O 函数。其他函数或函数系列需要特殊的操作系统支持，如 Winsows 或 Macintosh OS 提供的特殊图形界面。这些有针对性、非标准的函数让程序员能更有效地使用特定计算机编写程序。本章只着重讲解所有系统都通用的标准 I/O 函数，用这些函数编写的可移植程序很容易从一个系统移植到另一个系统。处理文件输入/输出的程序也可以使用这些函数。

许多程序都有输入验证，即判断用户的输入是否与程序期望的输入匹配。本章将演示一些与输入验证相关的问题和解决方案。

8.1 单字符 I/O：`getchar()`和`putchar()`

第 7 章中提到过，`getchar()`和`putchar()`每次只处理一个字符。你可能认为这种方法实在太笨拙了，毕竟与我们的阅读方式相差甚远。但是，这种方法很适合计算机。而且，这是绝大多数文本（即，普通文字）处理程序所用的核心方法。为了帮助读者回忆这些函数的工作方式，请看程序清单 8.1。该程序获取从键盘输入的字符，并把这些字符发送到屏幕上。程序使用 while 循环，当读到#字符时停止。

程序清单 8.1　`echo.c` 程序

```
/* echo.c -- 重复输入 */
#include <stdio.h>
int main(void)
{
```

```
    char ch;

    while ((ch = getchar()) != '#')
        putchar(ch);

    return 0;
}
```

自从 ANSI C 标准发布以后，C 就把 `stdio.h` 头文件与使用 `getchar()` 和 `putchar()` 相关联，这就是为什么程序中要包含这个头文件的原因（其实，`getchar()` 和 `putchar()` 都不是真正的函数，它们被定义为供预处理器使用的宏，我们在第 16 章中再详细讨论）。运行该程序后，与用户的交互如下：

```
Hello, there. I would[enter]
Hello, there. I would
like a #3 bag of potatoes.[enter]
like a
```

读者可能好奇，为何输入的字符能直接显示在屏幕上？如果用一个特殊字符（如，#）来结束输入，就无法在文本中使用这个字符，是否有更好的方法结束输入？要回答这些问题，首先要了解 C 程序如何处理键盘输入，尤其是缓冲和标准输入文件的概念。

8.2 缓冲区

如果在老式系统运行程序清单 8.1，你输入文本时可能显示如下：

```
HHeellllloo,, tthheerree.. II wwoouulldd[enter]

lliikkee aa #
```

以上行为是个例外。像这样回显用户输入的字符后立即重复打印该字符是属于*无缓冲*（或*直接*）输入，即正在等待的程序可立即使用输入的字符。对于该例，大部分系统在用户按下 Enter 键之前不会重复打印刚输入的字符，这种输入形式属于*缓冲*输入。用户输入的字符被收集并储存在一个被称为*缓冲区*（*buffer*）的临时存储区，按下 Enter 键后，程序才可使用用户输入的字符。图 8.1 比较了这两种输入。

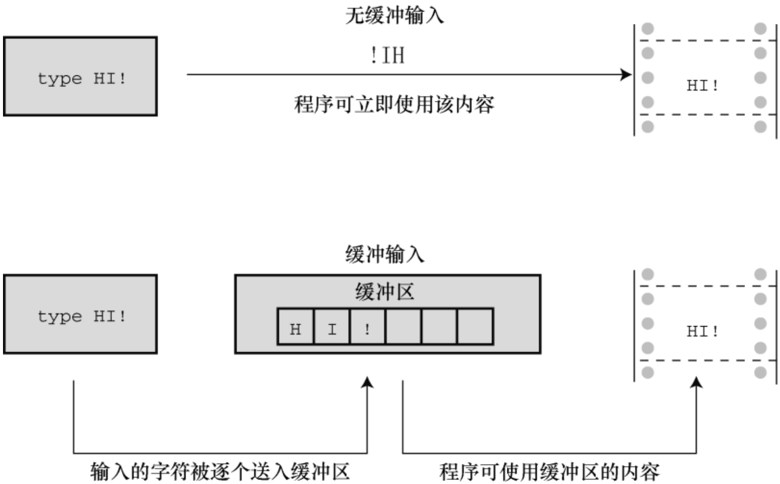


图 8.1　缓冲输入和无缓冲输入

为什么要有缓冲区？首先，把若干字符作为一个块进行传输比逐个发送这些字符节约时间。其次，如果用户打错字符，可以直接通过键盘修正错误。当最后按下 Enter 键时，传输的是正确的输入。

虽然缓冲输入好处很多，但是某些交互式程序也需要无缓冲输入。例如，在游戏中，你希望按下一个键就执行相应的指令。因此，缓冲输入和无缓冲输入都有用武之地。

缓冲分为两类：*完全缓冲 I/O* 和*行缓冲 I/O*。完全缓冲输入指的是当缓冲区被填满时才刷新缓冲区（内容被发送至目的地），通常出现在文件输入中。缓冲区的大小取决于系统，常见的大小是 512 字节和 4096 字节。行缓冲 I/O 指的是在出现换行符时刷新缓冲区。键盘输入通常是行缓冲输入，所以在按下 Enter 键后才刷新缓冲区。

那么，使用缓冲输入还是无缓冲输入？ANSI C 和后续的 C 标准都规定输入是缓冲的，不过最初 K&R 把这个决定权交给了编译器的编写者。读者可以运行 `echo.c` 程序观察输出的情况，了解所用的输出类型。

ANSI C 决定把缓冲输入作为标准的原因是：一些计算机不允许无缓冲输入。如果你的计算机允许无缓冲输入，那么你所用的 C 编译器很可能会提供一个无缓冲输入的选项。例如，许多 IBM PC 兼容机的编译器都为支持无缓冲输入提供一系列特殊的函数，其原型都在 `conio.h` 头文件中。这些函数包括用于回显无缓冲输入的 `getche()` 函数和用于无回显无缓冲输入的 `getch()` 函数（*回显输入*意味着用户输入的字符直接显示在屏幕上，*无回显输入*意味着击键后对应的字符不显示）。UNIX 系统使用另一种不同的方式控制缓冲。在 UNIX 系统中，可以使用 `ioctl()` 函数（该函数属于 UNIX 库，但是不属于 C 标准）指定待输入的类型，然后用 `getchar()` 执行相应的操作。在 ANSI C 中，用 `setbuf()` 和 `setvbuf()` 函数（详见第 13 章）控制缓冲，但是受限于一些系统的内部设置，这些函数可能不起作用。总之，ANSI 没有提供调用无缓冲输入的标准方式，这意味着是否能进行无缓冲输入取决于计算机系统。在这里要对使用无缓冲输入的朋友说声抱歉，本书假设所有的输入都是缓冲输入。

8.3 结束键盘输入

在 `echo.c` 程序（程序清单 8.1）中，只要输入的字符中不含#，那么程序在读到#时才会结束。但是，#也是一个普通的字符，有时不可避免要用到。应该用一个在文本中用不到的字符来标记输入完成，这样的字符不会无意间出现在输入中，在你不希望结束程序的时候终止程序。C 的确提供了这样的字符，不过在此之前，先来了解一下 C 处理文件的方式。

8.3.1 文件、流和键盘输入

文件（*file*）是存储器中储存信息的区域。通常，文件都保存在某种永久存储器中（如，硬盘、U 盘或 DVD 等）。毫无疑问，文件对于计算机系统相当重要。例如，你编写的 C 程序就保存在文件中，用来编译 C 程序的程序也保存在文件中。后者说明，某些程序需要访问指定的文件。当编译储存在名为 `echo.c` 文件中的程序时，编译器打开 `echo.c` 文件并读取其中的内容。当编译器处理完后，会关闭该文件。其他程序，例如文字处理器，不仅要打开、读取和关闭文件，还要把数据写入文件。

C 是一门强大、灵活的语言，有许多用于打开、读取、写入和关闭文件的库函数。从较低层面上，C 可以使用主机操作系统的基本文件工具直接处理文件，这些直接调用操作系统的函数被称为*底层 I/O*（*low-level I/O*）。由于计算机系统各不相同，所以不可能为普通的底层 I/O 函数创建标准库，ANSI C 也不打算这样做。然而从较高层面上，C 还可以通过*标准 I/O 包*（*standard I/O package*）来处理文件。这涉及创建用于处理文件的标准模型和一套标准 I/O 函数。在这一层面上，具体的 C 实现负责处理不同系统的差异，以便用户使用统一的界面。

上面讨论的差异指的是什么？例如，不同的系统储存文件的方式不同。有些系统把文件的内容储存在一处，而文件相关的信息储存在另一处；有些系统在文件中创建一份文件描述。在处理文件方面，有些系统使用单个换行符标记行末尾，而其他系统可能使用回车符和换行符的组合来表示行末尾。有些系统用最

小字节来衡量文件的大小，有些系统则以字节块的大小来衡量。

如果使用标准 I/O 包，就不用考虑这些差异。因此，可以用 `if (ch == '\n')`检查换行符。即使系统实际用的是回车符和换行符的组合来标记行末尾，I/O 函数会在两种表示法之间相互转换。

从概念上看，C 程序处理的是流而不是直接处理文件。流（*stream*）是一个实际输入或输出映射的理想化数据流。这意味着不同属性和不同种类的输入，由属性更统一的流来表示。于是，打开文件的过程就是把流与文件相关联，而且读写都通过流来完成。

第 13 章将更详细地讨论文件。本章着重理解 C 把输入和输出设备视为存储设备上的普通文件，尤其是把键盘和显示设备视为每个 C 程序自动打开的文件。`stdin` 流表示键盘输入，`stdout` 流表示屏幕输出。`getchar()`、`putchar()`、`printf()`和 `scanf()`函数都是标准 I/O 包的成员，处理这两个流。

以上讨论的内容说明，可以用处理文件的方式来处理键盘输入。例如，程序读文件时要能检测文件的末尾才知道应在何处停止。因此，C 的输入函数内置了文件结尾检测器。既然可以把键盘输入视为文件，那么也应该能使用文件结尾检测器结束键盘输入。下面我们从文件开始，学习如何结束文件。

8.3.2　文件结尾

计算机操作系统要以某种方式判断文件的开始和结束。检测文件结尾的一种方法是，在文件末尾放一个特殊的字符标记文件结尾。CP/M、IBM-DOS 和 MS-DOS 的文本文件曾经用过这种方法。如今，这些操作系统可以使用内嵌的 Ctrl+Z 字符来标记文件结尾。这曾经是操作系统使用的唯一标记，不过现在有一些其他的选择，例如记录文件的大小。所以现代的文本文件不一定有嵌入的 Ctrl+Z，但是如果有，该操作系统会将其视为一个文件结尾标记。图 8.2 演示了这种方法。

散文原文：

Ishphat the robot
slid open the hatch
and shouted his challenge.

文件中的散文：

```
Ishphat the robot\n slid open the hatch\n and shouted his challenge.\n^Z
```

图 8.2　带文件结尾标记的文件

操作系统使用的另一种方法是储存文件大小的信息。如果文件有 3000 字节，程序在读到 3000 字节时便达到文件的末尾。MS-DOS 及其相关系统使用这种方法处理二进制文件，因为用这种方法可以在文件中储存所有的字符，包括 Ctrl+Z。新版的 DOS 也使用这种方法处理文本文件。UNIX 使用这种方法处理所有的文件。

无论操作系统实际使用何种方法检测文件结尾，在 C 语言中，用 `getchar()`读取文件检测到文件结尾时将返回一个特殊的值，即 `EOF`（end of file 的缩写）。`scanf()`函数检测到文件结尾时也返回 `EOF`。通常，`EOF` 定义在 `stdio.h` 文件中：

```
#define EOF (-1)
```

为什么是`-1`？因为 `getchar()`函数的返回值通常都介于 `0`～`127`，这些值对应标准字符集。但是，如果系统能识别扩展字符集，该函数的返回值可能在 `0`～`255` 之间。无论哪种情况，`-1` 都不对应任何字符，所以，该值可用于标记文件结尾。

某些系统也许把 `EOF` 定义为`-1` 以外的值，但是定义的值一定与输入字符所产生的返回值不同。如果包

含 stdio.h 文件，并使用 EOF 符号，就不必担心 EOF 值不同的问题。这里关键要理解 EOF 是一个值，标志着检测到文件结尾，并不是在文件中找得到的符号。

那么，如何在程序中使用 EOF？把 getchar()的返回值和 EOF 作比较。如果两值不同，就说明没有到达文件结尾。也就是说，可以使用下面这样的表达式：

```
while ((ch = getchar()) != EOF)
```

如果正在读取的是键盘输入不是文件会怎样？绝大部分系统（不是全部）都有办法通过键盘模拟文件结尾条件。了解这些以后，读者可以重写程序清单 8.1 的程序，如程序清单 8.2 所示。

程序清单 8.2　echo_eof.c 程序

```
/* echo_eof.c -- 重复输入，直到文件结尾 */
#include <stdio.h>
int main(void)
{
    int ch;

    while ((ch = getchar()) != EOF)
        putchar(ch);

    return 0;
}
```

注意下面几点。

- 不用定义 EOF，因为 stdio.h 中已经定义过了。
- 不用担心 EOF 的实际值，因为 EOF 在 stdio.h 中用#define 预处理指令定义，可直接使用，不必再编写代码假定 EOF 为某值。
- 变量 ch 的类型从 char 变为 int，因为 char 类型的变量只能表示 0～255 的无符号整数，但是 EOF 的值是-1。还好，getchar()函数实际返回值的类型是 int，所以它可以读取 EOF 字符。如果实现使用有符号的 char 类型，也可以把 ch 声明为 char 类型，但最好还是用更通用的形式。
- 由于 getchar()函数的返回类型是 int，如果把 getchar()的返回值赋给 char 类型的变量，一些编译器会警告可能丢失数据。
- ch 是整数不会影响 putchar()，该函数仍然会打印等价的字符。
- 使用该程序进行键盘输入，要设法输入 EOF 字符。不能只输入字符 EOF，也不能只输入-1（输入-1 会传送两个字符：一个连字符和一个数字 1）。正确的方法是，必须找出当前系统的要求。例如，在大多数 UNIX 和 Linux 系统中，在一行开始处按下 Ctrl+D 会传输文件结尾信号。许多微型计算机系统都把一行开始处的 Ctrl+Z 识别为文件结尾信号，一些系统把任意位置的 Ctrl+Z 解释成文件结尾信号。

下面是在 UNIX 系统下运行 echo_eof.c 程序的缓冲示例：

```
She walks in beauty, like the night
She walks in beauty, like the night
  Of cloudless climes and starry skies...
  Of cloudless climes and starry skies...
                        Lord Byron
                        Lord Byron
[Ctrl+D]
```

每次按下 Enter 键，系统便会处理缓冲区中储存的字符，并在下一行打印该输入行的副本。这个过程一直持续到以 UNIX 风格模拟文件结尾（按下 Ctrl+D）。在 PC 中，要按下 Ctrl+Z。

我们暂停一会。既然 echo_eof.c 程序能把用户输入的内容拷贝到屏幕上，那么考虑一下该程序还可以做什么。假设以某种方式把一个文件传送给它，然后它把文件中的内容打印在屏幕上，当到达文件结尾发现 EOF 信号时停止。或者，假设以某种方式把程序的输出定向到一个文件，然后通过键盘输入数据，用 echo_eof.c 来储存在文件中输入的内容。假设同时使用这两种方法：把输入从一个文件定向到 echo_eof.c 中，并把输出发送至另一个文件，然后便可以使用 echo_eof.c 来拷贝文件。这个小程序有查看文件内容、创建一个新文件、拷贝文件的潜力，没想到一个小程序竟然如此多才多艺！关键是要控制输入流和输出流，这是我们下一个要讨论的主题。

注意　模拟 EOF 和图形界面

模拟 EOF 的概念是在使用文本界面的命令行环境中产生的。在这种环境中，用户通过击键与程序交互，由操作系统生成 EOF 信号。但是在一些实际应用中，却不能很好地转换成图形界面（如 Windows 和 Macintosh），这些用户界面包含更复杂的鼠标移动和按钮点击。程序要模拟 EOF 的行为依赖于编译器和项目类型。例如，Ctrl+Z 可以结束输入或整个程序，这取决于特定的设置。

8.4　重定向和文件

输入和输出涉及函数、数据和设备。例如，考虑 echo_eof.c，该程序使用输入函数 getchar()。输出设备（我们假设）是键盘，输入数据流由字符组成。假设你希望输入函数和数据类型不变，仅改变程序查找数据的位置。那么，程序如何知道去哪里查找输入？

在默认情况下，C 程序使用标准 I/O 包查找标准输入作为输入源。这就是前面介绍过的 stdin 流，它是把数据读入计算机的常用方式。它可以是一个过时的设备，如磁带、穿孔卡或电传打印机，或者（假设）是键盘，甚至是一些先进技术，如语音输入。然而，现代计算机非常灵活，可以让它到别处查找输入。尤其是，可以让一个程序从文件中查找输入，而不是从键盘。

程序可以通过两种方式使用文件。第 1 种方法是，显式使用特定的函数打开文件、关闭文件、读取文件、写入文件，诸如此类。我们在第 13 章中再详细介绍这种方法。第 2 种方法是，设计能与键盘和屏幕互动的程序，通过不同的渠道重定向输入至文件和从文件输出。换言之，把 stdin 流重新赋给文件。继续使用 getchar() 函数从输入流中获取数据，但它并不关心从流的什么位置获取数据。虽然这种重定向的方法在某些方面有些限制，但是用起来比较简单，而且能让读者熟悉普通的文件处理技术。

重定向的一个主要问题与操作系统有关，与 C 无关。尽管如此，许多 C 环境中（包括 UNIX、Linux 和 Windows 命令提示模式）都有重定向特性，而且一些 C 实现还在某些缺乏重定向特性的系统中模拟它。在 UNIX 上运行苹果 OS X，可以用 UNIX 命令行模式启动 Terminal 应用程序。接下来我们介绍 UNIX、Linux 和 Windows 的重定向。

8.4.1　UNIX、Linux 和 DOS 重定向

UNIX（运行命令行模式时）、Linux（ditto）和 Window 命令行提示（模仿旧式 DOS 命令行环境）都能重定向输入、输出。重定向输入让程序使用文件而不是键盘来输入，重定向输出让程序输出至文件而不是屏幕。

1. 重定向输入

假设已经编译了 echo_eof.c 程序，并把可执行版本放入一个名为 echo_eof（或者在 Windows 系统中名为 echo_eof.exe）的文件中。运行该程序，输入可执行文件名：

```
echo_eof
```

该程序的运行情况和前面描述的一样，获取用户从键盘输入的输入。现在，假设你要用该程序处理名为 words 的文本文件。文本文件（*text file*）是内含文本的文件，其中储存的数据是我们可识别的字符。文件的内容可以是一篇散文或者 C 程序。内含机器语言指令的文件（如储存可执行程序的文件）不是文本文件。由于该程序的操作对象是字符，所以要使用文本文件。只需用下面的命令代替上面的命令即可：

```
echo_eof < words
```

<符号是 UNIX 和 DOS/Windows 的重定向运算符。该运算符使 words 文件与 stdin 流相关联，把文件中的内容导入 echo_eof 程序。echo_eof 程序本身并不知道（或不关心）输入的内容是来自文件还是键盘，它只知道这是需要导入的字符流，所以它读取这些内容并把字符逐个打印在屏幕上，直至读到文件结尾。因为 C 把文件和 I/O 设备放在一个层面，所以文件就是现在的 I/O 设备。试试看！

> **注意　重定向**
>
> 对于 UNIX、Linux 和 Windows 命令提示，<两侧的空格是可选的。一些系统，如 AmigaDOS（那些喜欢怀旧的人使用的系统），支持重定向，但是在重定向符号和文件名之间不允许有空格。

下面是一个特殊的 words 文件的运行示例，$是 UNIX 和 Linux 的标准提示符。在 Windows/DOS 系统中见到的 DOS 提示可能是 A>或 C>。

```
$ echo_eof < words
The world is too much with us: late and soon,
Getting and spending, we lay waste our powers:
Little we see in Nature that is ours;
We have given our hearts away, a sordid boon!
$
```

2. 重定向输出

现在假设要用 echo_eof 把键盘输入的内容发送到名为 mywords 的文件中。然后，输入以下命令并开始输入：

```
echo_eof>mywords
```

>符号是第 2 个重定向运算符。它创建了一个名为 mywords 的新文件，然后把 echo_eof 的输出（即，你输入字符的副本）重定向至该文件中。重定向把 stdout 从显示设备（即，显示器）赋给 mywords 文件。如果已经有一个名为 mywords 的文件，通常会擦除该文件的内容，然后替换新的内容（但是，许多操作系统有保护现有文件的选项，使其成为只读文件）。所有出现在屏幕的字母都是你刚才输入的，其副本储存在文件中。在下一行的开始处按下 Ctrl+D（UNIX）或 Ctrl+Z（DOS）即可结束该程序。如果不知道输入什么内容，可参照下面的示例。这里，我们使用 UNIX 提示符$。记住在每行的末尾单击 Enter 键，这样才能把缓冲区的内容发送给程序。

```
$ echo_eof > mywords
You should have no problem recalling which redirection
operator does what. Just remember that each operator points
in the direction the information flows. Think of it as
a funnel.
[Ctrl+D]
$
```

按下 Ctrl+D 或 Ctrl+Z 后，程序会结束，你的系统会提示返回。程序是否起作用了？UNIX 的 `ls` 命令或 Windows 命令行提示模式的 `dir` 命令可以列出文件名，会显示 `mywords` 文件已存在。可以使用 UNIX 或 Linux 的 `cat` 或 DOS 的 `type` 命令检查文件中的内容，或者再次使用 `echo_eof`，这次把文件重定向到程序：

```
$ echo_eof < mywords
You should have no problem recalling which redirection
operator does what. Just remember that each operator points
in the direction the information flows. Think of it as a
funnel.
$
```

3. 组合重定向

现在，假设你希望制作一份 `mywords` 文件的副本，并命名为 `savewords`。只需输入以下命令即可：

```
echo_eof < mywords > savewords
```

下面的命令也起作用，因为命令与重定向运算符的顺序无关：

```
echo_eof > savewords < mywords
```

注意：在一条命令中，输入文件名和输出文件名不能相同。

```
echo_eof < mywords > mywords....<--错误
```

原因是`> mywords` 在输入之前已导致原 `mywords` 的长度被截断为 0。

总之，在 UNIX、Linux 或 Windows/DOS 系统中使用两个重定向运算符（<和>）时，要遵循以下原则。

- 重定向运算符连接一个可执行程序（包括标准操作系统命令）和一个数据文件，不能用于连接一个数据文件和另一个数据文件，也不能用于连接一个程序和另一个程序。
- 使用重定向运算符不能读取多个文件的输入，也不能把输出定向至多个文件。
- 通常，文件名和运算符之间的空格不是必须的，除非是偶尔在 UNIX shell、Linux shell 或 Windows 命令行提示模式中使用的有特殊含义的字符。例如，我们用过的 `echo_eof<words`。

以上介绍的都是正确的例子，下面来看一下错误的例子，`addup` 和 `count` 是两个可执行程序，`fish` 和 `beets` 是两个文本文件：

```
fish > beets              ←违反第 1 条规则
addup < count             ←违反第 1 条规则
addup < fish < beets      ←违反第 2 条规则
count > beets fish        ←违反第 2 条规则
```

UNIX、Linux 或 Windows/DOS 还有>>运算符，该运算符可以把数据添加到现有文件的末尾，而 | 运算符能把一个文件的输出连接到另一个文件的输入。欲了解所有相关运算符的内容，请参阅 UNIX 的相关书籍，如 *UNIX Primer Plus，Third Edition*（Wilson、Pierce 和 Wessler 合著）。

4. 注释

重定位让你能使用键盘输入程序文件。要完成这一任务，程序要测试文件的末尾。例如，第 7 章演示的统计单词程序（程序清单 7.7），计算单词个数直至遇到第 1 个|字符。把 `ch` 的 `char` 类型改成 `int` 类型，把循环测试中的 | 替换成 `EOF`，便可用该程序来计算文本文件中的单词量。

重定向是一个命令行概念，因为我们要在命令行输入特殊的符号发出指令。如果不使用命令行环境，也可以使用重定向。首先，一些集成开发环境提供了菜单选项，让用户指定重定向。其次，对于 Windows 系统，可以打开命令提示窗口，并在命令行运行可执行文件。Microsoft Visual Studio 的默认设置是把可执行文件放在项目文件夹的子文件夹，称为 Debug。文件名和项目名的基本名相同，文件名的扩展名为.exe。

默认情况下，Xcode 在给项目命名后才能命名可执行文件，并将其放在 Debug 文件夹中。在 UNIX 系统中，可以通过 Terminal 工具运行可执行文件。从使用上看，Terminal 比命令行编译器（GCC 或 Clang）简单。

如果用不了重定向，可以用程序直接打开文件。程序清单 8.3 演示了一个注释较少的示例。我们学到第 13 章时再详细讲解。待读取的文件应该与可执行文件位于同一目录。

程序清单 8.3 file_eof.c 程序

```
// file_eof.c --打开一个文件并显示该文件
#include <stdio.h>
#include <stdlib.h>             // 为了使用 exit()
int main()
{
    int ch;
    FILE * fp;
    char fname[50];             // 储存文件名

    printf("Enter the name of the file: ");
    scanf("%s", fname);
    fp = fopen(fname, "r");     // 打开待读取文件
    if (fp == NULL)             // 如果失败
    {
        printf("Failed to open file. Bye\n");
        exit(1);                // 退出程序
    }
    // getc(fp)从打开的文件中获取一个字符
    while ((ch = getc(fp)) != EOF)
        putchar(ch);
    fclose(fp);                 // 关闭文件

    return 0;
}
```

小结：如何重定向输入和输出

绝大部分 C 系统都可以使用重定向，可以通过操作系统重定向所有程序，或只在 C 编译器允许的情况下重定向 C 程序。假设 prog 是可执行程序名，file1 和 file2 是文件名。

把输出重定向至文件：>

```
prog >file1
```

把输入重定向至文件：<

```
prog <file2
```

组合重定向：

```
prog <file2 >file1
prog >file1 <file2
```

这两种形式都是把 file2 作为输入、file1 作为输出。

留白：

一些系统要求重定向运算符左侧有一个空格，右侧没有空格。而其他系统（如，UNIX）允许在重定位运算符两侧有空格或没有空格。

8.5　创建更友好的用户界面

大部分人偶尔会写一些中看不中用的程序。还好，C 提供了大量工具让输入更顺畅，处理过程更顺利。不过，学习这些工具会导致新的问题。本节的目标是，指导读者解决这些问题并创建更友好的用户界面，让交互数据输入更方便，减少错误输入的影响。

8.5.1　使用缓冲输入

缓冲输入用起来比较方便，因为在把输入发送给程序之前，用户可以编辑输入。但是，在使用输入的字符时，它也会给程序员带来麻烦。前面示例中看到的问题是，缓冲输入要求用户按下 Enter 键发送输入。这一动作也传送了换行符，程序必须妥善处理这个麻烦的换行符。我们以一个猜谜程序为例。用户选择一个数字，程序猜用户选中的数字是多少。该程序使用的方法单调乏味，先不要在意算法，我们关注的重点在输入和输出。查看程序清单 8.4，这是猜谜程序的最初版本，后面我们会改进。

程序清单 8.4　`guess.c` **程序**

```
/* guess.c -- 一个拖沓且错误的猜数字程序 */
#include <stdio.h>
int main(void)
{
    int guess = 1;

    printf("Pick an integer from 1 to 100. I will try to guess ");
    printf("it.\nRespond with a y if my guess is right and with");
    printf("\nan n if it is wrong.\n");
    printf("Uh...is your number %d?\n", guess);
    while (getchar() != 'y')      /* 获取响应，与 y 做对比 */
        printf("Well, then, is it %d?\n", ++guess);
    printf("I knew I could do it!\n");

    return 0;
}
```

下面是程序的运行示例：

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
Well, then, is it 3?
n
Well, then, is it 4?
Well, then, is it 5?
y
I knew I could do it!
```

撇开这个程序糟糕的算法不谈，我们先选择一个数字。注意，每次输入 `n` 时，程序打印了两条消息。这是由于程序读取 `n` 作为用户否定了数字 `1`，然后还读取了一个换行符作为用户否定了数字 `2`。

一种解决方案是，使用 `while` 循环丢弃输入行最后剩余的内容，包括换行符。这种方法的优点是，能把 `no` 和 `no way` 这样的响应视为简单的 `n`。程序清单 8.4 的版本会把 `no` 当作两个响应。下面用循环修正

这个问题：

```
while (getchar() != 'y')   /* 获取响应，与 y 做对比*/
{
    printf("Well, then, is it %d?\n", ++guess);
    while (getchar() != '\n')
        continue;          /* 跳过剩余的输入行 */
}
```

使用以上循环后，该程序的输出示例如下：

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
no
Well, then, is it 3?
no sir
Well, then, is it 4?
forget it
Well, then, is it 5?
y
I knew I could do it!
```

这的确是解决了换行符的问题。但是，该程序还是会把 f 被视为 n。我们用 if 语句筛选其他响应。首先，添加一个 char 类型的变量储存响应：

```
char response;
```

修改后的循环如下：

```
while ((response = getchar()) != 'y') /* 获取响应 */
{
    if (response == 'n')
        printf("Well, then, is it %d?\n", ++guess);
    else
        printf("Sorry, I understand only y or n.\n");
    while (getchar() != '\n')
        continue; /* 跳过剩余的输入行 */
}
```

现在，程序的运行示例如下：

```
Pick an integer from 1 to 100. I will try to guess it.
Respond with a y if my guess is right and with
an n if it is wrong.
Uh...is your number 1?
n
Well, then, is it 2?
no
Well, then, is it 3?
no sir
Well, then, is it 4?
forget it
Sorry, I understand only y or n.
n
Well, then, is it 5?
y
I knew I could do it!
```

在编写交互式程序时，应该事先预料到用户可能会输入错误，然后设计程序处理用户的错误输入。在用户出错时通知用户再次输入。

当然，无论你的提示写得多么清楚，总会有人误解，然后抱怨这个程序设计得多么糟糕。

8.5.2　混合数值和字符输入

假设程序要求用 getchar()处理字符输入，用 scanf()处理数值输入，这两个函数都能很好地完成任务，但是不能把它们混用。因为 getchar()读取每个字符，包括空格、制表符和换行符；而 scanf()在读取数字时则会跳过空格、制表符和换行符。

我们通过程序清单 8.5 来解释这种情况导致的问题。该程序读入一个字符和两个数字，然后根据输入的两个数字指定的行数和列数打印该字符。

程序清单 8.5　showchar1.c 程序

```
/* showchar1.c -- 有较大 I/O 问题的程序 */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;                 /* 待打印字符    */
    int rows, cols;         /* 行数和列数 */
    printf("Enter a character and two integers:\n");
    while ((ch = getchar()) != '\n')
    {
        scanf("%d %d", &rows, &cols);
        display(ch, rows, cols);
        printf("Enter another character and two integers;\n");
        printf("Enter a newline to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}

void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n');/* 结束一行并开始新的一行 */
    }
}
```

注意，该程序以 int 类型读取字符（这样做可以检测 EOF），但是却以 char 类型把字符传递给 display()函数。因为 char 比 int 小，一些编译器会给出类型转换的警告。可以忽略这些警告，或者用下面的强制类型转换消除警告：

```
display(char(ch), rows, cols);
```

在该程序中，main()负责获取数据，display()函数负责打印数据。下面是该程序的一个运行示例，

看看有什么问题：

```
Enter a character and two integers:
c 2 3
ccc
ccc
Enter another character and two integers;
Enter a newline to quit.
Bye.
```

该程序开始时运行良好。你输入 c 2 3，程序打印 c 字符 2 行 3 列。然后，程序提示输入第 2 组数据，还没等你输入数据程序就退出了！这是什么情况？又是换行符在捣乱，这次是输入行中紧跟在 3 后面的换行符。scanf()函数把这个换行符留在输入队列中。和 scanf()不同，getchar()不会跳过换行符，所以在进入下一轮迭代时，你还没来得及输入字符，它就读取了换行符，然后将其赋给 ch。而 ch 是换行符正式终止循环的条件。

要解决这个问题，程序要跳过一轮输入结束与下一轮输入开始之间的所有换行符或空格。另外，如果该程序不在 getchar()测试时，而在 scanf()阶段终止程序会更好。修改后的版本如程序清单 8.6 所示。

程序清单 8.6　showchar2.c 程序

```
/* showchar2.c -- 按指定的行列打印字符 */
#include <stdio.h>
void display(char cr, int lines, int width);
int main(void)
{
    int ch;                /* 待打印字符*/
    int rows, cols;        /* 行数和列数 */

    printf("Enter a character and two integers:\n");
    while ((ch = getchar()) != '\n')
    {
        if (scanf("%d %d", &rows, &cols) != 2)
            break;
        display(ch, rows, cols);
        while (getchar() != '\n')
            continue;
        printf("Enter another character and two integers;\n");
        printf("Enter a newline to quit.\n");
    }
    printf("Bye.\n");

    return 0;
}

void display(char cr, int lines, int width)
{
    int row, col;

    for (row = 1; row <= lines; row++)
    {
        for (col = 1; col <= width; col++)
            putchar(cr);
        putchar('\n');     /* 结束一行并开始新的一行 */
    }
}
```

while 循环实现了丢弃 scanf()输入后面所有字符（包括换行符）的功能，为循环的下一轮读取做好了准备。该程序的运行示例如下：

```
Enter a character and two integers:
c 1 2
cc
Enter another character and two integers;
Enter a newline to quit.
! 3 6
!!!!!!
!!!!!!
!!!!!!
Enter another character and two integers;
Enter a newline to quit.

Bye.
```

在 if 语句中使用一个 break 语句，可以在 scanf()的返回值不等于 2 时终止程序，即如果一个或两个输入值不是整数或者遇到文件结尾就终止程序。

8.6　输入验证

在实际应用中，用户不一定会按照程序的指令行事。用户的输入和程序期望的输入不匹配时常发生，这会导致程序运行失败。作为程序员，除了完成编程的本职工作，还要事先预料一些可能的输入错误，这样才能编写出能检测并处理这些问题的程序。

例如，假设你编写了一个处理非负数整数的循环，但是用户很可能输入一个负数。你可以使用关系表达式来排除这种情况：

```
long n;
scanf("%ld", &n);      // 获取第 1 个值
while (n >= 0)         // 检测不在范围内的值
{
    // 处理 n
    scanf("%ld", &n);  // 获取下一个值
}
```

另一类潜在的陷阱是，用户可能输入错误类型的值，如字符 q。排除这种情况的一种方法是，检查 scanf()的返回值。回忆一下，scanf()返回成功读取项的个数。因此，下面的表达式当且仅当用户输入一个整数时才为真：

```
scanf("%ld", &n) == 1
```

结合上面的 while 循环，可改进为：

```
long n;
while (scanf("%ld", &n) == 1 && n >= 0)
{
    // 处理 n
}
```

while 循环条件可以描述为“当输入是一个整数且该整数为正时”。

对于最后的例子，当用户输入错误类型的值时，程序结束。然而，也可以让程序友好些，提示用户再次输入正确类型的值。在这种情况下，要处理有问题的输入。如果 scanf()没有成功读取，就会将其留在输入队列中。这里要明确，输入实际上是字符流。可以使用 getchar()函数逐字符地读取输入，甚至可以把这些想法都结合在一个函数中，如下所示：

```
long get_long(void)
{
    long input;
    char ch;
    while (scanf("%ld", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch); // 处理错误的输入
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}
```

该函数要把一个 int 类型的值读入变量 input 中。如果读取失败，函数则进入外层 while 循环体。然后内层循环逐字符地读取错误的输入。注意，该函数丢弃该输入行的所有剩余内容。还有一个方法是，只丢弃下一个字符或单词，然后该函数提示用户再次输入。外层循环重复运行，直到用户成功输入整数，此时 scanf()的返回值为 1。

在用户输入整数后，程序可以检查该值是否有效。考虑一个例子，要求用户输入一个上限和一个下限来定义值的范围。在该例中，你可能希望程序检查第 1 个值是否大于第 2 个值（通常假设第 1 个值是较小的那个值），除此之外还要检查这些值是否在允许的范围内。例如，当前的档案查找一般不会接受 1958 年以前和 2014 年以后的查询任务。这个限制可以在一个函数中实现。

假设程序中包含了 stdbool.h 头文件。如果当前系统不允许使用_Bool，把 bool 替换成 int，把 true 替换成 1，把 false 替换成 0 即可。注意，如果输入无效，该函数返回 true，所以函数名为 bad_limits()：

```
bool bad_limits(long begin, long end,long low, long high)
{
    bool not_good = false;
    if (begin > end)
    {
        printf("%ld isn't smaller than %ld.\n", begin, end);
        not_good = true;
    }
    if (begin < low || end < low)
    {
        printf("Values must be %ld or greater.\n", low);
        not_good = true;
    }
    if (begin > high || end > high)
    {
        printf("Values must be %ld or less.\n", high);
        not_good = true;
    }

    return not_good;
}
```

程序清单 8.7 使用了上面的两个函数为一个进行算术运算的函数提供整数，该函数计算特定范围内所有整数的平方和。程序限制了范围的上限是 10000000，下限是-10000000。

程序清单 8.7　checking.c 程序

```
// checking.c -- 输入验证
#include <stdio.h>
#include <stdbool.h>
// 验证输入是一个整数
long get_long(void);
// 验证范围的上下限是否有效
bool bad_limits(long begin, long end,
                long low, long high);
// 计算a~b之间的整数平方和
double sum_squares(long a, long b);
int main(void)
{
    const long MIN = -10000000L;    // 范围的下限
    const long MAX = +10000000L;    // 范围的上限
    long start;                     // 用户指定的范围最小值
    long stop;                      // 用户指定的范围最大值
    double answer;

    printf("This program computes the sum of the squares of "
           "integers in a range.\nThe lower bound should not "
           "be less than -10000000 and\nthe upper bound "
           "should not be more than +10000000.\nEnter the "
           "limits (enter 0 for both limits to quit):\n"
           "lower limit: ");
    start = get_long();
    printf("upper limit: ");
    stop = get_long();
    while (start != 0 || stop != 0)
    {
        if (bad_limits(start, stop, MIN, MAX))
            printf("Please try again.\n");
        else
        {
            answer = sum_squares(start, stop);
            printf("The sum of the squares of the integers ");
            printf("from %ld to %ld is %g\n",
                   start, stop, answer);
        }
        printf("Enter the limits (enter 0 for both "
               "limits to quit):\n");
        printf("lower limit: ");
        start = get_long();
        printf("upper limit: ");
        stop = get_long();
    }
    printf("Done.\n");

    return 0;
}

long get_long(void)
{
    long input;
    char ch;
```

```
    while (scanf("%ld", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch);          // 处理错误输入
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}

double sum_squares(long a, long b)
{
    double total = 0;
    long i;

    for (i = a; i <= b; i++)
        total += (double) i * (double) i;

    return total;
}

bool bad_limits(long begin, long end,
    long low, long high)
{
    bool not_good = false;

    if (begin > end)
    {
        printf("%ld isn't smaller than %ld.\n", begin, end);
        not_good = true;
    }
    if (begin < low || end < low)
    {
        printf("Values must be %ld or greater.\n", low);
        not_good = true;
    }
    if (begin > high || end > high)
    {
        printf("Values must be %ld or less.\n", high);
        not_good = true;
    }

    return not_good;
}
```

下面是该程序的输出示例：

```
This program computes the sum of the squares of integers in a range.
The lower bound should not be less than -10000000 and
the upper bound should not be more than +10000000.
Enter the limits (enter 0 for both limits to quit):
lower limit: low
low is not an integer.
Please enter an integer value, such as 25, -178, or 3: 3
```

```
upper limit: a big number
a big number is not an integer.
Please enter an integer value, such as 25, -178, or 3: 12
The sum of the squares of the integers from 3 to 12 is 645
Enter the limits (enter 0 for both limits to quit):
lower limit: 80
upper limit: 10
80 isn't smaller than 10.
Please try again.
Enter the limits (enter 0 for both limits to quit):
lower limit: 0
upper limit: 0
Done.
```

8.6.1　分析程序

虽然 checking.c 程序的核心计算部分（sum_squares()函数）很短，但是输入验证部分比以往程序示例要复杂。接下来分析其中的一些要素，先着重讨论程序的整体结构。

程序遵循模块化的编程思想，使用独立函数（模块）来验证输入和管理显示。程序越大，使用模块化编程就越重要。

main()函数管理程序流，为其他函数委派任务。它使用 get_long()获取值、while 循环处理值、badlimits()函数检查值是否有效、sum_squres()函数处理实际的计算：

```
start = get_long();
printf("upper limit: ");
stop = get_long();
while (start != 0 || stop != 0)
{
    if (bad_limits(start, stop, MIN, MAX))
        printf("Please try again.\n");
    else
    {
        answer = sum_squares(start, stop);
        printf("The sum of the squares of the integers ");
        printf("from %ld to %ld is %g\n", start, stop, answer);
    }
    printf("Enter the limits (enter 0 for both "
        "limits to quit):\n");
    printf("lower limit: ");
    start = get_long();
    printf("upper limit: ");
    stop = get_long();
}
```

8.6.2　输入流和数字

在编写处理错误输入的代码时（如程序清单 8.7），应该很清楚 C 是如何处理输入的。考虑下面的输入：

```
is 28 12.4
```

在我们眼中，这就像是一个由字符、整数和浮点数组成的字符串。但是对 C 程序而言，这是一个字节流。第 1 个字节是字母 i 的字符编码，第 2 个字节是字母 s 的字符编码，第 3 个字节是空格字符的字符编码，第 4 个字节是数字 2 的字符编码，等等。所以，如果 get_long()函数处理这一行输入，第 1 个字符是非数字，那么整行输入都会被丢弃，包括其中的数字，因为这些数字只是该输入行中的其他字符：

```
while ((ch = getchar()) != '\n')
    putchar(ch);  // 处理错误的输入
```

虽然输入流由字符组成，但是也可以设置 scanf()函数把它们转换成数值。例如，考虑下面的输入：

42

如果在 scanf()函数中使用%c 转换说明，它只会读取字符 4 并将其储存在 char 类型的变量中。如果使用%s 转换说明，它会读取字符 4 和字符 2 这两个字符，并将其储存在字符数组中。如果使用%d 转换说明，scanf()同样会读取两个字符，但是随后会计算出它们对应的整数值：4×10+2，即 42，然后将表示该整数的二进制数储存在 int 类型的变量中。如果使用%f 转换说明，scanf()也会读取两个字符，计算出它们对应的数值 42.0，用内部的浮点表示法表示该值，并将结果储存在 float 类型的变量中。

简而言之，输入由字符组成，但是 scanf()可以把输入转换成整数值或浮点数值。使用转换说明（如%d 或%f）限制了可接受输入的字符类型，而 getchar()和使用%c 的 scanf()接受所有的字符。

8.7　菜单浏览

许多计算机程序都把菜单作为用户界面的一部分。菜单给用户提供方便的同时，却给程序员带来了一些麻烦。我们看看其中涉及了哪些问题。

菜单给用户提供了一份响应程序的选项。假设有下面一个例子：

```
Enter the letter of your choice:
a. advice          b. bell
c. count           q. quit
```

理想状态是，用户输入程序所列选项之一，然后程序根据用户所选项完成任务。作为一名程序员，自然希望这一过程能顺利进行。因此，第 1 个目标是：当用户遵循指令时程序顺利运行；第 2 个目标是：当用户没有遵循指令时，程序也能顺利运行。显而易见，要实现第 2 个目标难度较大，因为很难预料用户在使用程序时的所有错误情况。

现在的应用程序通常使用图形界面，可以点击按钮、查看对话框、触摸图标，而不是我们示例中的命令行模式。但是，两者的处理过程大致相同：给用户提供选项、检查并执行用户的响应、保护程序不受误操作的影响。除了界面不同，它们底层的程序结构也几乎相同。但是，使用图形界面更容易通过限制选项控制输入。

8.7.1　任务

我们来更具体地分析一个菜单程序需要执行哪些任务。它要获取用户的响应，根据响应选择要执行的动作。另外，程序应该提供返回菜单的选项。C 的 switch 语句是根据选项决定行为的好工具，用户的每个选择都可以对应一个特定的 case 标签。使用 while 语句可以实现重复访问菜单的功能。因此，我们写出以下伪代码：

```
获取选项
当选项不是'q'时
    转至相应的选项并执行
    获取下一个选项
```

8.7.2　使执行更顺利

当你决定实现这个程序时，就要开始考虑如何让程序顺利运行（顺利运行指的是，处理正确输入和错误输入时都能顺利运行）。例如，你能做的是让“获取选项”部分的代码筛选掉不合适的响应，只把正确的

响应传入 switch。这表明需要为输入过程提供一个只返回正确响应的函数。结合 while 循环和 switch 语句，其程序结构如下：

```
#include <stdio.h>
char get_choice(void);
void count(void);
int main(void)
{
    int choice;

    while ((choice = get_choice()) != 'q')
    {
        switch (choice)
        {
            case 'a': printf("Buy low, sell high.\n");
                      break;
            case 'b': putchar('\a'); /* ANSI */
                      break;
            case 'c': count();
                      break;
            default:  printf("Program error!\n");
                      break;
        }
    }
    return 0;
}
```

定义 get_choice()函数只能返回'a'、'b'、'c'和'q'。get_choice()的用法和 getchar()相同，两个函数都是获取一个值，并与终止值（该例中是'q'）作比较。我们尽量简化实际的菜单选项，以便读者把注意力集中在程序结构上。稍后再讨论 count()函数。default 语句可以方便调试。如果 get_choice()函数没能把返回值限制为菜单指定的几个选项值，default 语句有助于发现问题所在。

get_choice()函数

下面的伪代码是设计这个函数的一种方案：

显示选项

获取用户的响应

当响应不合适时

　　提示用户再次输入

　　获取用户的响应

下面是一个简单而笨拙的实现：

```
char get_choice(void)
{
    int ch;
    printf("Enter the letter of your choice:\n");
    printf("a. advice          b. bell\n");
    printf("c. count           q. quit\n");
    ch = getchar();
    while ((ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = getchar();
```

```
    }
    return ch;
}
```

缓冲输入依旧带来些麻烦，程序把用户每次按下 Return 键产生的换行符视为错误响应。为了让程序的界面更流畅，该函数应该跳过这些换行符。

这类问题有多种解决方案。一种是用名为 get_first()的新函数替换 getchar()函数，读取一行的第 1 个字符并丢弃剩余的字符。这种方法的优点是，把类似 act 这样的输入视为简单的 a，而不是继续把 act 中的 c 作为选项 c 的一个有效的响应。我们重写输入函数如下：

```
char get_choice(void)
{
    int ch;
    printf("Enter the letter of your choice:\n");
    printf("a. advice           b. bell\n");
    printf("c. count            q. quit\n");
    ch = get_first();
    while ((ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = getfirst();
    }
    return ch;
}
char get_first(void)
{
    int ch;
    ch = getchar();    /* 读取下一个字符 */
    while (getchar() != '\n')
        continue; /* 跳过该行剩下的内容 */
    return ch;
}
```

8.7.3　混合字符和数值输入

前面分析过混合字符和数值输入会产生一些问题，创建菜单也有这样的问题。例如，假设 count()函数（选择 c）的代码如下：

```
 void count(void)
{
    int n, i;
    printf("Count how far? Enter an integer:\n");
    scanf("%d", &n);
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
}
```

如果输入 3 作为响应，scanf()会读取 3 并把换行符留在输入队列中。下次调用 get_choice()将导致 get_first()返回这个换行符，从而导致我们不希望出现的行为。

重写 get_first()，使其返回下一个非空白字符而不仅仅是下一个字符，即可修复这个问题。我们把这个任务留给读者作为练习。另一种方法是，在 count()函数中清理换行符，如下所示：

```
void count(void)
{
    int n, i;
    printf("Count how far? Enter an integer:\n");
```

```
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while (getchar() != '\n')
        continue;
}
```

该函数借鉴了程序清单 8.7 中的 get_long()函数，将其改为 get_int()获取 int 类型的数据而不是 long 类型的数据。回忆一下，原来的 get_long()函数如何检查有效输入和让用户重新输入。程序清单 8.8 演示了菜单程序的最终版本。

程序清单 8.8　menuette.c **程序**

```
/* menuette.c -- 菜单程序 */
#include <stdio.h>
char get_choice(void);
char get_first(void);
int get_int(void);
void count(void);
int main(void)
{
    int choice;
    void count(void);

    while ((choice = get_choice()) != 'q')
    {
        switch (choice)
        {
            case 'a':  printf("Buy low, sell high.\n");
                       break;
            case 'b':  putchar('\a');   /* ANSI */
                       break;
            case 'c':  count();
                       break;
            default:   printf("Program error!\n");
                       break;
        }
    }
    printf("Bye.\n");

    return 0;
}

void count(void)
{
    int n, i;

    printf("Count how far? Enter an integer:\n");
    n = get_int();
    for (i = 1; i <= n; i++)
        printf("%d\n", i);
    while (getchar() != '\n')
        continue;
}

char get_choice(void)
{
```

```
    int ch;

    printf("Enter the letter of your choice:\n");
    printf("a. advice         b. bell\n");
    printf("c. count          q. quit\n");
    ch = get_first();
    while ((ch < 'a' || ch > 'c') && ch != 'q')
    {
        printf("Please respond with a, b, c, or q.\n");
        ch = get_first();
    }

    return ch;
}

char get_first(void)
{
    int ch;

    ch = getchar();
    while (getchar() != '\n')
        continue;

    return ch;
}

int get_int(void)
{
    int input;
    char ch;

    while (scanf("%d", &input) != 1)
    {
        while ((ch = getchar()) != '\n')
            putchar(ch);  // 处理错误输出
        printf(" is not an integer.\nPlease enter an ");
        printf("integer value, such as 25, -178, or 3: ");
    }

    return input;
}
```

下面是该程序的一个运行示例：

```
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
a
Buy low, sell high.
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
count
Count how far? Enter an integer:
two
two is not an integer.
```

```
Please enter an integer value, such as 25, -178, or 3: 5

1
2
3
4
5
Enter the letter of your choice:
a. advice           b. bell
c. count            q. quit
d
Please respond with a, b, c, or q.
q
```

要写出一个自己十分满意的菜单界面并不容易。但是，在开发了一种可行的方案后，可以在其他情况下复用这个菜单界面。

学完以上程序示例后，还要注意在处理较复杂的任务时，如何让函数把任务委派给另一个函数。这样让程序更模块化。

8.8　关键概念

C 程序把输入作为传入的字节流。getchar()函数把每个字符解释成一个字符编码。scanf()函数以同样的方式看待输入，但是根据转换说明，它可以把字符输入转换成数值。许多操作系统都提供重定向，允许用文件代替键盘输入，用文件代替显示器输出。

程序通常接受特殊形式的输入。可以在设计程序时考虑用户在输入时可能犯的错误，在输入验证部分处理这些错误情况，让程序更强健更友好。

对于一个小型程序，输入验证可能是代码中最复杂的部分。处理这类问题有多种方案。例如，如果用户输入错误类型的信息，可以终止程序，也可以给用户提供有限次或无限次机会重新输入。

8.9　本章小结

许多程序使用 getchar()逐字符读取输入。通常，系统使用行缓冲输入，即当用户按下 Enter 键后输入才被传送给程序。按下 Enter 键也传送了一个换行符，编程时要注意处理这个换行符。ANSI C 把缓冲输入作为标准。

通过标准 I/O 包中的一系列函数，以统一的方式处理不同系统中的不同文件形式，是 C 语言的特性之一。getchar()和 scanf()函数也属于这一系列。当检测到文件结尾时，这两个函数都返回 EOF（被定义在 stdio.h 头文件中）。在不同系统中模拟文件结尾条件的方式稍有不同。在 UNIX 系统中，在一行开始处按下 Ctrl+D 可以模拟文件结尾条件；而在 DOS 系统中则使用 Ctrl+Z。

许多操作系统（包括 UNIX 和 DOS）都有重定向的特性，因此可以用文件代替键盘和屏幕进行输入和输出。读到 EOF 即停止读取的程序可用于键盘输入和模拟文件结尾信号，或者用于重定向文件。

混合使用 getchar()和 scanf()时，如果在调用 getchar()之前，scanf()在输入行留下一个换行符，会导致一些问题。不过，意识到这个问题就可以在程序中妥善处理。

编写程序时，要认真设计用户界面。事先预料一些用户可能会犯的错误，然后设计程序妥善处理这些错误情况。

8.10 复习题

复习题的参考答案在附录 A 中。

1. putchar(getchar())是一个有效表达式，它实现什么功能？getchar(putchar())是否也是有效表达式？

2. 下面的语句分别完成什么任务？

   a．putchar('H');

   b．putchar('\007');

   c．putchar('\n');

   d．putchar('\b');

3. 假设有一个名为 count 的可执行程序，用于统计输入的字符数。设计一个使用 count 程序统计 essay 文件中字符数的命令行，并把统计结果保存在 essayct 文件中。

4. 给定复习题 3 中的程序和文件，下面哪一条是有效的命令？

   a．essayct <essay

   b．count essay

   c．essay >count

5. EOF 是什么？

6. 对于给定的输出（ch 是 int 类型，而且是缓冲输入），下面各程序段的输出分别是什么？

   a．输入如下：

   ```
   If you quit, I will.[enter]
   ```

   程序段如下：

   ```
   while ((ch = getchar()) != 'i')
      putchar(ch);
   ```

   b．输入如下：

   ```
   Harhar[enter]
   ```

   程序段如下：

   ```
   while ((ch = getchar()) != '\n')
   {
      putchar(ch++);
      putchar(++ch);
   }
   ```

7. C 如何处理不同计算机系统中的不同文件和换行约定？

8. 在使用缓冲输入的系统中，把数值和字符混合输入会遇到什么潜在的问题？

8.11 编程练习

下面的一些程序要求输入以 EOF 终止。如果你的操作系统很难或根本无法使用重定向，请使用一些其他的测试来终止输入，如读到&字符时停止。

1. 设计一个程序，统计在读到文件结尾之前读取的字符数。

2. 编写一个程序，在遇到 EOF 之前，把输入作为字符流读取。程序要打印每个输入的字符及其相应的 ASCII 十进制值。注意，在 ASCII 序列中，空格字符前面的字符都是非打印字符，要特殊处理

这些字符。如果非打印字符是换行符或制表符，则分别打印\n 或\t。否则，使用控制字符表示法。例如，ASCII 的 1 是 Ctrl+A，可显示为^A。注意，A 的 ASCII 值是 Ctrl+A 的值加上 64。其他非打印字符也有类似的关系。除每次遇到换行符打印新的一行之外，每行打印 10 对值。（注意：不同的操作系统其控制字符可能不同。）

3. 编写一个程序，在遇到 EOF 之前，把输入作为字符流读取。该程序要报告输入中的大写字母和小写字母的个数。假设大小写字母数值是连续的。或者使用 ctype.h 库中合适的分类函数更方便。

4. 编写一个程序，在遇到 EOF 之前，把输入作为字符流读取。该程序要报告平均每个单词的字母数。不要把空白统计为单词的字母。实际上，标点符号也不应该统计，但是现在暂时不同考虑这么多（如果你比较在意这点，考虑使用 ctype.h 系列中的 ispunct()函数）。

5. 修改程序清单 8.4 的猜数字程序，使用更智能的猜测策略。例如，程序最初猜 50，询问用户是猜大了、猜小了还是猜对了。如果猜小了，那么下一次猜测的值应是 50 和 100 中值，也就是 75。如果这次猜大了，那么下一次猜测的值应是 50 和 75 的中值，等等。使用二分查找（*binary search*）策略，如果用户没有欺骗程序，那么程序很快就会猜到正确的答案。

6. 修改程序清单 8.8 中的 get_first()函数，让该函数返回读取的第 1 个非空白字符，并在一个简单的程序中测试。

7. 修改第 7 章的编程练习 8，用字符代替数字标记菜单的选项。用 q 代替 5 作为结束输入的标记。

8. 编写一个程序，显示一个提供加法、减法、乘法、除法的菜单。获得用户选择的选项后，程序提示用户输入两个数字，然后执行用户刚才选择的操作。该程序只接受菜单提供的选项。程序使用 float 类型的变量储存用户输入的数字，如果用户输入失败，则允许再次输入。进行除法运算时，如果用户输入 0 作为第 2 个数（除数），程序应提示用户重新输入一个新值。该程序的一个运行示例如下：

```
Enter the operation of your choice:
a. add           s. subtract
m. multiply      d. divide
q. quit
a
Enter first number: 22 .4
Enter second number: one
one is not an number.
Please enter a number, such as 2.5, -1.78E8, or 3: 1
22.4 + 1 = 23.4
Enter the operation of your choice:
a. add           s. subtract
m. multiply      d. divide
q. quit
d
Enter first number: 18.4
Enter second number: 0
Enter a number other than 0: 0.2
18.4 / 0.2 = 92
Enter the operation of your choice:
a. add           s. subtract
m. multiply      d. divide
q. quit
q
Bye.
```

第9章 函数

本章介绍以下内容：

- 关键字：return
- 运算符：*（一元）、&（一元）
- 函数及其定义方式
- 如何使用参数和返回值
- 如何把指针变量用作函数参数
- 函数类型
- ANSI C 原型
- 递归

如何组织程序？C的设计思想是，把函数用作构件块。我们已经用过C标准库的函数，如printf()、scanf()、getchar()、putchar()和 strlen()。现在要进一步学习如何创建自己的函数。前面章节中已大致介绍了相关过程，本章将巩固以前学过的知识并做进一步的拓展。

9.1 复习函数

首先，什么是函数？函数（*function*）是完成特定任务的独立程序代码单元。语法规则定义了函数的结构和使用方式。虽然C中的函数和其他语言中的函数、子程序、过程作用相同，但是细节上略有不同。一些函数执行某些动作，如printf()把数据打印到屏幕上；一些函数找出一个值供程序使用，如strlen()把指定字符串的长度返回给程序。一般而言，函数可以同时具备以上两种功能。

为什么要使用函数？首先，使用函数可以省去编写重复代码的苦差。如果程序要多次完成某项任务，那么只需编写一个合适的函数，就可以在需要时使用这个函数，或者在不同的程序中使用该函数，就像许多程序中使用putchar()一样。其次，即使程序只完成某项任务一次，也值得使用函数。因为函数让程序更加模块化，从而提高了程序代码的可读性，更方便后期修改、完善。例如，假设要编写一个程序完成以下任务：

- 读入一系列数字；
- 分类这些数字；
- 找出这些数字的平均值；
- 打印一份柱状图。

可以使用下面的程序：

```
#include <stdio.h>
#define SIZE 50
int main(void)
{
```

```
    float list[SIZE];

    readlist(list, SIZE);
    sort(list, SIZE);
    average(list, SIZE);
    bargraph(list, SIZE);
    return 0;
}
```

当然，还要编写 4 个函数 readlist()、sort()、average()和 bargraph()的实现细节。描述性的函数名能清楚地表达函数的用途和组织结构。然后，单独设计和测试每个函数，直到函数都能正常完成任务。如果这些函数够通用，还可以用于其他程序。

许多程序员喜欢把函数看作是根据传入信息（输入）及其生成的值或响应的动作（输出）来定义的“黑盒”。如果不是自己编写函数，根本不用关心黑盒的内部行为。例如，使用 printf()时，只需知道给该函数传入格式字符串或一些参数以及 printf()生成的输出，无需了解 printf()的内部代码。以这种方式看待函数有助于把注意力集中在程序的整体设计，而不是函数的实现细节上。因此，在动手编写代码之前，仔细考虑一下函数应该完成什么任务，以及函数和程序整体的关系。

如何了解函数？首先要知道如何正确地定义函数、如何调用函数和如何建立函数间的通信。我们从一个简单的程序示例开始，帮助读者理清这些内容，然后再详细讲解。

9.1.1　创建并使用简单函数

我们的第 1 个目标是创建一个在一行打印 40 个星号的函数，并在一个打印表头的程序中使用该函数。如程序清单 9.1 所示，该程序由 main()和 starbar()组成。

程序清单 9.1　lethead1.c 程序

```
/* lethead1.c */
#include <stdio.h>
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40

void starbar(void);  /* 函数原型 */

int main(void)
{
    starbar();
    printf("%s\n", NAME);
    printf("%s\n", ADDRESS);
    printf("%s\n", PLACE);
    starbar();      /* 使用函数 */

    return 0;
}

void starbar(void)   /* 定义函数    */
{
    int count;

    for (count = 1; count <= WIDTH; count++)
        putchar('*');
```

```
    putchar('\n');
}
```

该程序的输出如下：

```
****************************************
GIGATHINK, INC.
101 Megabuck Plaza
Megapolis, CA 94904
****************************************
```

9.1.2 分析程序

该程序要注意以下几点。

- 程序在 3 处使用了 `starbar` 标识符：*函数原型*（*function prototype*）告诉编译器函数 `starbar()` 的类型；*函数调用*（*function call*）表明在此处执行函数；*函数定义*（*function definition*）明确地指定了函数要做什么。
- 函数和变量一样，有多种类型。任何程序在使用函数之前都要声明该函数的类型。因此，在 `main()` 函数定义的前面出现了下面的 ANSI C 风格的函数原型：

  ```
  void starbar(void);
  ```

  圆括号表明 `starbar` 是一个函数名。第 1 个 `void` 是函数类型，`void` 类型表明函数没有返回值。第 2 个 `void`（在圆括号中）表明该函数不带参数。分号表明这是在声明函数，不是定义函数。也就是说，这行声明了程序将使用一个名为 `starbar()`、没有返回值、没有参数的函数，并告诉编译器在别处查找该函数的定义。对于不识别 ANSI C 风格原型的编译器，只需声明函数的类型，如下所示：

  ```
  void starbar();
  ```

  注意，一些老版本的编译器甚至连 `void` 都识别不了。如果使用这种编译器，就要把没有返回值的函数声明为 `int` 类型。当然，最好还是换一个新的编译器。
- 一般而言，函数原型指明了函数的返回值类型和函数接受的参数类型。这些信息称为该函数的*签名*（*signature*）。对于 `starbar()` 函数而言，其签名是该函数没有返回值，没有参数。
- 程序把 `starbar()` 原型置于 `main()` 的前面。当然，也可以放在 `main()` 里面的声明变量处。放在哪个位置都可以。
- 在 `main()` 中，执行到下面的语句时调用了 `starbar()` 函数：

  ```
  starbar();
  ```

  这是调用 `void` 类型函数的一种形式。当计算机执行到 `starbar();` 语句时，会找到该函数的定义并执行其中的内容。执行完 `starbar()` 中的代码后，计算机返回*主调函数*（*calling function*）继续执行下一行（本例中，主调函数是 `main()`），见图 9.1（更确切地说，编译器把 C 程序翻译成执行以上操作的机器语言代码）。
- 程序中 `strarbar()` 和 `main()` 的定义形式相同。首先函数头包括函数类型、函数名和圆括号，接着是左花括号、变量声明、函数表达式语句，最后以右花括号结束（见图 9.2）。注意，函数头中的 `starbar()` 后面没有分号，告诉编译器这是定义 `starbar()`，而不是调用函数或声明函数原型。
- 程序把 `starbar()` 和 `main()` 放在一个文件中。当然，也可以把它们分别放在两个文件中。把函数都放在一个文件中的单文件形式比较容易编译，而使用多个文件方便在不同的程序中使用同一个

函数。如果把函数放在一个单独的文件中，要把#define 和#include 指令也放入该文件。我们稍后会讨论使用多个文件的情况。现在，先把所有的函数都放在一个文件中。main()的右花括号告诉编译器该函数结束的位置，后面的 starbar()函数头告诉编译器 starbar()是一个函数。

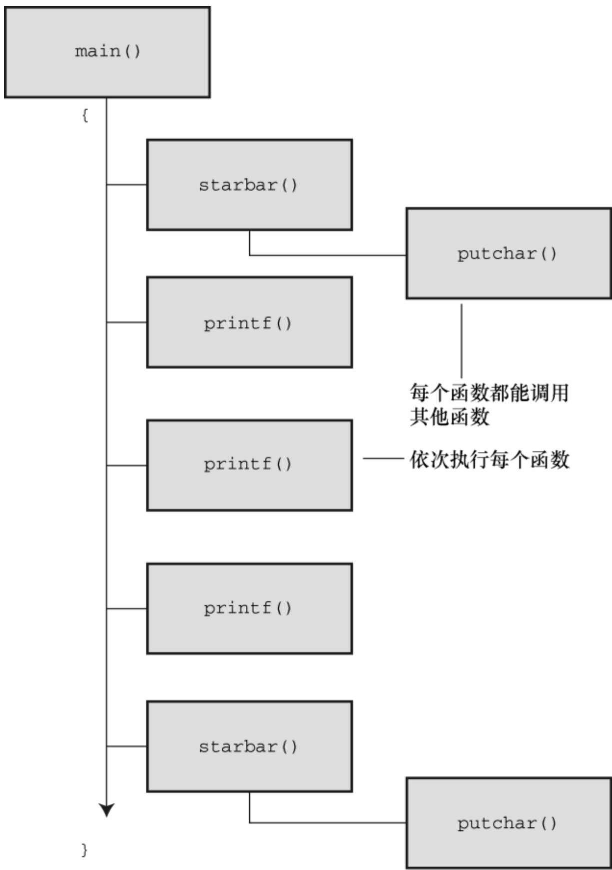


图 9.1　lethead1.c（程序清单 9.1）的程序流

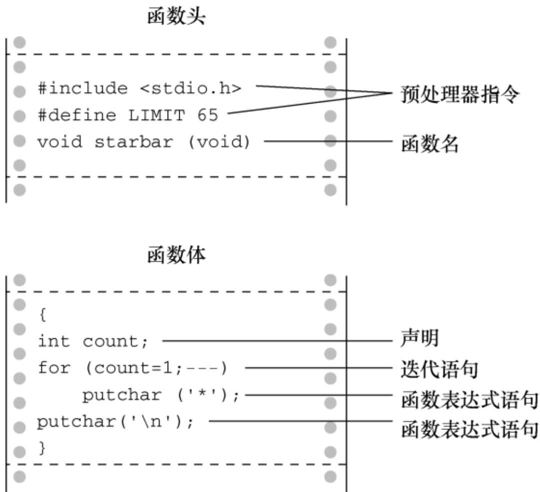


图 9.2　简单函数的结构

■ starbar()函数中的变量 count 是局部变量（*local variable*），意思是该变量只属于 starbar()函数。可以在程序中的其他地方（包括 main()中）使用 count，这不会引起名称冲突，它们是同名的不同变量。

如果把 starbar()看作是一个黑盒，那么它的行为是打印一行星号。不用给该函数提供任何输入，因为调用它不需要其他信息。而且，它没有返回值，所以也不给 main()提供（或返回）任何信息。简而言之，starbar()不需要与主调函数通信。

接下来介绍一个函数间需要通信的例子。

9.1.3 函数参数

在程序清单 9.1 的输出中，如果文字能居中，信头会更加美观。可以通过在打印文字之前打印一定数量的空格来实现，这和打印一定数量的星号（starbar()函数）类似，只不过现在要打印的是一定数量的空格。虽然这是两个任务，但是任务非常相似，与其分别为它们编写一个函数，不如写一个更通用的函数，可以在两种情况下使用。我们设计一个新的函数 show_n_char()（显示一个字符 n 次）。唯一要改变的是使用内置的值来显示字符和重复的次数，show_n_char()将使用函数参数来传递这些值。

我们来具体分析。假设可用的空间是 40 个字符宽。调用 show_n_char('*', 40)应该正好打印一行 40 个星号，就像 starbar()之前做的那样。第 2 行 GIGATHINK, INT.的空格怎么处理？GIGATHINK, INT.是 15 个字符宽，所以第 1 个版本中，文字后面有 25 个空格。为了让文字居中，文字的左侧应该有 12 个空格，右侧有 13 个空格。因此，可以调用 show_n_char('*', 12)。

show_n_char()与 starbar()很相似，但是 show_n_char()带有参数。从功能上看，前者不会添加换行符，而后者会，因为 show_n_char()要把空格和文本打印成一行。程序清单 9.2 是修改后的版本。为强调参数的工作原理，程序使用了不同的参数形式。

程序清单 9.2 lethead2.c **程序**

```
/* lethead2.c */
#include <stdio.h>
#include <string.h>          /* 为 strlen()提供原型 */
#define NAME "GIGATHINK, INC."
#define ADDRESS "101 Megabuck Plaza"
#define PLACE "Megapolis, CA 94904"
#define WIDTH 40
#define SPACE ' '

void show_n_char(char ch, int num);

int main(void)
{
    int spaces;

    show_n_char('*', WIDTH);                    /* 用符号常量作为参数 */
    putchar('\n');
    show_n_char(SPACE, 12);                     /* 用符号常量作为参数 */
    printf("%s\n", NAME);
    spaces = (WIDTH - strlen(ADDRESS)) / 2; /* 计算要跳过多少个空格*/

    show_n_char(SPACE, spaces);                 /* 用一个变量作为参数*/
    printf("%s\n", ADDRESS);
```

```
    show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);

    printf("%s\n", PLACE);                    /* 用一个表达式作为参数    */
    show_n_char('*', WIDTH);
    putchar('\n');

    return 0;
}

/* show_n_char()函数的定义 */
void show_n_char(char ch, int num)
{
    int count;

    for (count = 1; count <= num; count++)
        putchar(ch);
}
```

该函数的运行结果如下：

```
****************************************
          GIGATHINK, INC.
        101 Megabuck Plaza
       Megapolis, CA 94904
****************************************
```

下面我们回顾一下如何编写一个带参数的函数，然后介绍这种函数的用法。

9.1.4　定义带形式参数的函数

函数定义从下面的 ANSI C 风格的函数头开始：

```
void show_n_char(char ch, int num)
```

该行告知编译器 show_n_char()使用两个参数 ch 和 num，ch 是 char 类型，num 是 int 类型。这两个变量被称为*形式参数*（*formal argument*，但是最近的标准推荐使用 *formal parameter*），简称形参。和定义在函数中变量一样，形式参数也是局部变量，属该函数私有。这意味着在其他函数中使用同名变量不会引起名称冲突。每次调用函数，就会给这些变量赋值。

注意，ANSI C 要求在每个变量前都声明其类型。也就是说，不能像普通变量声明那样使用同一类型的变量列表：

```
void dibs(int x, y, z)          /* 无效的函数头 */
void dubs(int x, int y, int z) /* 有效的函数头 */
```

ANSI C 也接受 ANSI C 之前的形式，但是将其视为废弃不用的形式：

```
void show_n_char(ch, num)
char ch;
int num;
```

这里，圆括号中只有参数名列表，而参数的类型在后面声明。注意，普通的局部变量在左花括号之后声明，而上面的变量在函数左花括号之前声明。如果变量是同一类型，这种形式可以用逗号分隔变量名列表，如下所示：

```
void dibs(x, y, z)
int x, y, z;        /* 有效 */
```

当前的标准正逐渐淘汰 ANSI 之前的形式。读者应对此有所了解，以便能看懂以前编写的程序，但是

自己编写程序时应使用现在的标准形式（C99 和 C11 标准继续警告这些过时的用法即将被淘汰）。

虽然 show_n_char()接受来自 main()的值，但是它没有返回值。因此，show_n_char()的类型是 void。

下面，我们来学习如何使用函数。

9.1.5 声明带形式参数函数的原型

在使用函数之前，要用 ANSI C 形式声明函数原型：

```
void show_n_char(char ch, int num);
```

当函数接受参数时，函数原型用逗号分隔的列表指明参数的数量和类型。根据个人喜好，你也可以省略变量名：

```
void show_n_char(char, int);
```

在原型中使用变量名并没有实际创建变量，char 仅代表了一个 char 类型的变量，以此类推。

再次提醒读者注意，ANSI C 也接受过去的声明函数形式，即圆括号内没有参数列表：

```
void show_n_char();
```

这种形式最终会从标准中剔除。即使没有被剔除，现在函数原型的设计也更有优势（稍后会介绍）。了解这种形式的写法是为了以后读得懂以前写的代码。

9.1.6 调用带实际参数的函数

在函数调用中，实际参数（*actual argument*，简称实参）提供了 ch 和 num 的值。考虑程序清单 9.2 中第 1 次调用 show_n_char()：

```
show_n_char(SPACE, 12);
```

实际参数是空格字符和 12。这两个值被赋给 show_n_char()中相应的形式参数：变量 ch 和 num。简而言之，形式参数是被调函数（*called function*）中的变量，实际参数是主调函数（*calling function*）赋给被调函数的具体值。如上例所示，实际参数可以是常量、变量，或甚至是更复杂的表达式。无论实际参数是何种形式都要被求值，然后该值被拷贝给被调函数相应的形式参数。以程序清单 9.2 中最后一次调用 show_n_char()为例：

```
show_n_char(SPACE, (WIDTH - strlen(PLACE)) / 2);
```

构成该函数第 2 个实际参数的是一个很长的表达式，对该表达式求值为 10。然后，10 被赋给变量 num。被调函数不知道也不关心传入的数值是来自常量、变量还是一般表达式。再次强调，实际参数是具体的值，该值要被赋给作为形式参数的变量（见图 9.3）。因为被调函数使用的值是从主调函数中拷贝而来，所以无论被调函数对拷贝数据进行什么操作，都不会影响主调函数中的原始数据。

> **注意　实际参数和形式参数**
>
> 实际参数是出现在函数调用圆括号中的表达式。形式参数是函数定义的函数头中声明的变量。调用函数时，创建了声明为形式参数的变量并初始化为实际参数的求值结果。程序清单 9.2 中，'*'和 WIDTH 都是第 1 次调用 show_n_char()时的实际参数，而 SPACE 和 11 是第 2 次调用 show_n_char()时的实际参数。在函数定义中，ch 和 num 都是该函数的形式参数。



图 9.3　形式参数和实际参数

9.1.7　黑盒视角

从黑盒的视角看 `show_n_char()`，待显示的字符和显示的次数是输入。执行后的结果是打印指定数量的字符。输入以参数的形式被传递给函数。这些信息清楚地表明了如何在 main()中使用该函数。而且，这也可以作为编写该函数的设计说明。

黑盒方法的核心部分是：`ch`、`num` 和 `count` 都是 `show_n_char()`私有的局部变量。如果在 `main()`中使用同名变量，那么它们相互独立，互不影响。也就是说，如果 `main()`有一个 `count` 变量，那么改变它的值不会改变 `show_n_char()`中的 `count`，反之亦然。黑盒里发生了什么对主调函数是不可见的。

9.1.8　使用 `return` 从函数中返回值

前面介绍了如何把信息从主调函数传递给被调函数。反过来，函数的返回值可以把信息从被调函数传回主调函数。为进一步说明，我们将创建一个返回两个参数中较小值的函数。由于函数被设计用来处理 `int` 类型的值，所以被命名为 `imin()`。另外，还要创建一个简单的 `main()`，用于检查 `imin()`是否正常工作。这种被设计用于测试函数的程序有时被称为*驱动程序*（*driver*），该驱动程序调用一个函数。如果函数成功通过了测试，就可以安装在一个更重要的程序中使用。程序清单 9.3 演示了这个驱动程序和返回最小值的函数。

程序清单 9.3　`lesser.c` 程序

```
/* lesser.c -- 找出两个整数中较小的一个 */
#include <stdio.h>
int imin(int, int);

int main(void)
{
    int evil1, evil2;

    printf("Enter a pair of integers (q to quit):\n");
```

```
    while (scanf("%d %d", &evil1, &evil2) == 2)
    {
        printf("The lesser of %d and %d is %d.\n",
               evil1, evil2, imin(evil1, evil2));
        printf("Enter a pair of integers (q to quit):\n");
    }
    printf("Bye.\n");

    return 0;
}

int imin(int n, int m)
{
    int min;

    if (n < m)
        min = n;
    else
        min = m;

    return min;
}
```

回忆一下，scanf()返回成功读数据的个数，所以如果输入不是两个整数会导致循环终止。下面是一个运行示例：

```
Enter a pair of integers (q to quit):
509 333
The lesser of 509 and 333 is 333.
Enter a pair of integers (q to quit):
-9393 6
The lesser of -9393 and 6 is -9393.
Enter a pair of integers (q to quit):
q
Bye.
```

关键字 return 后面的表达式的值就是函数的返回值。在该例中，该函数返回的值就是变量 min 的值。因为 min 是 int 类型的变量，所以 imin()函数的类型也是 int。

变量 min 属于 imin()函数私有，但是 return 语句把 min 的值传回了主调函数。下面这条语句的作用是把 min 的值赋给 lesser：

```
lesser = imin(n,m);
```

是否能像写成下面这样：

```
imin(n,m);
lesser = min;
```

不能。因为主调函数甚至不知道 min 的存在。记住，imin()中的变量是 imin()的局部变量。函数调用 imin(evil1, evil2)只是把两个变量的值拷贝了一份。

返回值不仅可以赋给变量，也可以被用作表达式的一部分。例如，可以这样：

```
answer = 2 * imin(z, zstar) + 25;
printf("%d\n", imin(-32 + answer, LIMIT));
```

返回值不一定是变量的值，也可以是任意表达式的值。例如，可以用以下的代码简化程序示例：

```
/* 返回最小值的函数，第 2 个版本 */
imin(int n,int m)
```

```
{
    return (n < m) ? n : m;
}
```

条件表达式的值是 n 和 m 中的较小者，该值要被返回给主调函数。虽然这里不要求用圆括号把返回值括起来，但是如果想让程序条理更清楚或统一风格，可以把返回值放在圆括号内。

如果函数返回值的类型与函数声明的类型不匹配会怎样？

```
int what_if(int n)
{
    double z = 100.0 / (double) n;
    return z; // 会发生什么？
}
```

实际得到的返回值相当于把函数中指定的返回值赋给与函数类型相同的变量所得到的值。因此在本例中，相当于把 z 的值赋给 int 类型的变量，然后返回 int 类型变量的值。例如，假设有下面的函数调用：

```
result = what_if(64);
```

虽然在 what_if()函数中赋给 z 的值是 1.5625，但是 return 语句返回确实 int 类型的值 1。

使用 return 语句的另一个作用是，终止函数并把控制返回给主调函数的下一条语句。因此，可以这样编写 imin()：

```
/*返回最小值的函数，第 3 个版本*/
imin(int n,int m)
{
    if (n < m)
        return n;
    else
        return m;
}
```

许多 C 程序员都认为只在函数末尾使用一次 return 语句比较好，因为这样做更方便浏览程序的人理解函数的控制流。但是，在函数中使用多个 return 语句也没有错。无论如何，对用户而言，这 3 个版本的函数用起来都一样，因为所有的输入和输出都完全相同，不同的是函数内部的实现细节。下面的版本也没问题：

```
/*返回最小值的函数，第 4 个版本*/
imin(int n, int m)
{
    if (n < m)
        return n;
    else
        return m;
    printf("Professor Fleppard is like totally a fopdoodle.\n");
}
```

return 语句导致 printf()语句永远不会被执行。如果 Fleppard 教授在自己的程序中使用这个版本的函数，可能永远不知道编写这个函数的学生对他的看法。

另外，还可以这样使用 return：

```
return;
```

这条语句会导致终止函数，并把控制返回给主调函数。因为 return 后面没有任何表达式，所以没有返回值，只有在 void 函数中才会用到这种形式。

9.1.9　函数类型

声明函数时必须声明函数的类型。带返回值的函数类型应该与其返回值类型相同，而没有返回值的函数应

声明为 void 类型。如果没有声明函数的类型，旧版本的 C 编译器会假定函数的类型是 int。这一惯例源于 C 的早期，那时的函数绝大多数都是 int 类型。然而，C99 标准不再支持 int 类型函数的这种假定设置。

类型声明是函数定义的一部分。要记住，函数类型指的是返回值的类型，不是函数参数的类型。例如，下面的函数头定义了一个带两个 int 类型参数的函数，但是其返回值是 double 类型。

```
double klink(int a, int b)
```

要正确地使用函数，程序在第 1 次使用函数之前必须知道函数的类型。方法之一是，把完整的函数定义放在第 1 次调用函数的前面。然而，这种方法增加了程序的阅读难度。而且，要使用的函数可能在 C 库或其他文件中。因此，通常的做法是提前声明函数，把函数的信息告知编译器。例如，程序清单 9.3 中的 main()函数包含以下几行代码：

```
#include <stdio.h>
int imin(int, int);
int main(void)
{
    int evil1, evil2, lesser;
```

第 2 行代码说明 imin 是一个函数名，有两个 int 类型的形参，且返回 int 类型的值。现在，编译器在程序中调用 imin()函数时就知道应该如何处理。

在程序清单 9.3 中，我们把函数的前置声明放在主调函数外面。当然，也可以放在主调函数里面。例如，重写 lesser.c（程序清单 9.3）的开头部分：

```
#include <stdio.h>
int main(void)
{
    int imin(int, int); /* 声明imin()函数的原型*/
    int evil1, evil2, lesser;
```

注意在这两种情况中，函数原型都声明在使用函数之前。

ANSI C 标准库中，函数被分成多个系列，每一系列都有各自的头文件。这些头文件中除了其他内容，还包含了本系列所有函数的声明。例如，stdio.h 头文件包含了标准 I/O 库函数（如，printf()和 scanf()）的声明。math.h 头文件包含了各种数学函数的声明。例如，下面的声明：

```
double sqrt(double);
```

告知编译器 sqrt()函数有一个 double 类型的形参，而且返回 double 类型的值。不要混淆函数的声明和定义。函数声明告知编译器函数的类型，而函数定义则提供实际的代码。在程序中包含 math.h 头文件告知编译器：sqrt()返回 double 类型，但是 sqrt()函数的代码在另一个库函数的文件中。

9.2　ANSI C 函数原型

在 ANSI C 标准之前，声明函数的方案有缺陷，因为只需要声明函数的类型，不用声明任何参数。下面我们看一下使用旧式的函数声明会导致什么问题。

下面是 ANSI 之前的函数声明，告知编译器 imin()返回 int 类型的值：

```
int imin();
```

然而，以上函数声明并未给出 imin()函数的参数个数和类型。因此，如果调用 imin()时使用的参数个数不对或类型不匹配，编译器根本不会察觉出来。

9.2.1　问题所在

我们看看与 imax()函数相关的一些示例，该函数与 imin()函数关系密切。程序清单 9.4 演示了一个

程序，用过去声明函数的方式声明了 imax()函数，然后错误地使用该函数。

程序清单 9.4　misuse.c **程序**

```
/* misuse.c -- 错误地使用函数 */
#include <stdio.h>
int imax();      /* 旧式函数声明 */

int main(void)
{
    printf("The maximum of %d and %d is %d.\n",3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(n, m)
int n, m;
{
    return (n > m ? n : m);
}
```

第 1 次调用 printf()时省略了 imax()的一个参数，第 2 次调用 printf()时用两个浮点参数而不是整数参数。尽管有些问题，但程序可以编译和运行。

下面是使用 Xcode 4.6 运行的输出示例：

```
The maximum of 3 and 5 is 1606416656.
The maximum of 3 and 5 is 3886.
```

使用 gcc 运行该程序，输出的值是 1359379472 和 1359377160。这两个编译器都运行正常，之所以输出错误的结果，是因为它们运行的程序没有使用函数原型。

到底是哪里出了问题？由于不同系统的内部机制不同，所以出现问题的具体情况也不同。下面介绍的是使用 PC 和 VAX 的情况。主调函数把它的参数储存在被称为栈（*stack*）的临时存储区，被调函数从栈中读取这些参数。对于该例，这两个过程并未相互协调。主调函数根据函数调用中的实际参数决定传递的类型，而被调函数根据它的形式参数读取值。因此，函数调用 imax(3)把一个整数放在栈中。当 imax()函数开始执行时，它从栈中读取两个整数。而实际上栈中只存放了一个待读取的整数，所以读取的第 2 个值是当时恰好在栈中的其他值。

第 2 次使用 imax()函数时，它传递的是 float 类型的值。这次把两个 double 类型的值放在栈中（回忆一下，当 float 类型被作为参数传递时会被升级为 double 类型）。在我们的系统中，两个 double 类型的值就是两个 64 位的值，所以 128 位的数据被放在栈中。当 imax()从栈中读取两个 int 类型的值时，它从栈中读取前 64 位。在我们的系统中，每个 int 类型的变量占用 32 位。这些数据对应两个整数，其中较大的是 3886。

9.2.2　ANSI 的解决方案

针对参数不匹配的问题，ANSI C 标准要求在函数声明时还要声明变量的类型，即使用*函数原型*（*function prototype*）来声明函数的返回类型、参数的数量和每个参数的类型。未标明 imax()函数有两个 int 类型的参数，可以使用下面两种函数原型来声明：

```
int imax(int, int);
int imax(int a, int b);
```

第 1 种形式使用以逗号分隔的类型列表，第 2 种形式在类型后面添加了变量名。注意，这里的变量名

是假名，不必与函数定义的形式参数名一致。

有了这些信息，编译器可以检查函数调用是否与函数原型匹配。参数的数量是否正确？参数的类型是否匹配？以 imax()为例，如果两个参数都是数字，但是类型不匹配，编译器会把实际参数的类型转换成形式参数的类型。例如，imax(3.0, 5.0)会被转换成 imax(3, 5)。我们用函数原型替换程序清单 9.4 中的函数声明，如程序清单 9.5 所示。

程序清单 9.5 proto.c 程序

```
/* proto.c -- 使用函数原型 */
#include <stdio.h>
int imax(int, int);       /* 函数原型 */
int main(void)
{
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3));
    printf("The maximum of %d and %d is %d.\n",
            3, 5, imax(3.0, 5.0));
    return 0;
}

int imax(int n, int m)
{
    return (n > m ? n : m);
}
```

编译程序清单 9.5 时，我们的编译器给出调用的 imax()函数参数太少的错误消息。

如果是类型不匹配会怎样？为探索这个问题，我们用 imax(3, 5)替换 imax(3)，然后再次编译该程序。这次编译器没有给出任何错误信息，程序的输出如下：

```
The maximum of 3 and 5 is 5.
The maximum of 3 and 5 is 5.
```

如上文所述，第 2 次调用中的 3.0 和 5.0 被转换成 3 和 5，以便函数能正确地处理输入。

虽然没有错误消息，但是我们的编译器还是给出了警告：double 转换成 int 可能会导致丢失数据。例如，下面的函数调用：

```
imax(3.9, 5.4)
```

相当于：

```
imax(3, 5)
```

错误和警告的区别是：错误导致无法编译，而警告仍然允许编译。一些编译器在进行类似的类型转换时不会通知用户，因为 C 标准中对此未作要求。不过，许多编译器都允许用户选择警告级别来控制编译器在描述警告时的详细程度。

9.2.3 无参数和未指定参数

假设有下面的函数原型：

```
void print_name();
```

一个支持 ANSI C 的编译器会假定用户没有用函数原型来声明函数，它将不会检查参数。为了表明函数确实没有参数，应该在圆括号中使用 void 关键字：

```
void print_name(void);
```

支持 ANSI C 的编译器解释为 print_name() 不接受任何参数。然后在调用该函数时，编译器会检查以确保没有使用参数。

一些函数接受（如，printf()和 scanf()）许多参数。例如对于 printf()，第 1 个参数是字符串，但是其余参数的类型和数量都不固定。对于这种情况，ANSI C 允许使用部分原型。例如，对于 printf()可以使用下面的原型：

```
int printf(const char *, ...);
```

这种原型表明，第 1 个参数是一个字符串（第 11 章中将详细介绍），可能还有其他未指定的参数。

C 库通过 stdarg.h 头文件提供了一个定义这类（形参数量不固定的）函数的标准方法。第 16 章中详细介绍相关内容。

9.2.4　函数原型的优点

函数原型是 C 语言的一个强有力的工具，它让编译器捕获在使用函数时可能出现的许多错误或疏漏。如果编译器没有发现这些问题，就很难觉察出来。是否必须使用函数原型？不一定。你也可以使用旧式的函数声明（即不用声明任何形参），但是这样做的弊大于利。

有一种方法可以省略函数原型却保留函数原型的优点。首先要明白，之所以使用函数原型，是为了让编译器在第 1 次执行到该函数之前就知道如何使用它。因此，把整个函数定义放在第 1 次调用该函数之前，也有相同的效果。此时，函数定义也相当于函数原型。对于较小的函数，这种用法很普遍：

```
// 下面这行代码既是函数定义，也是函数原型
int imax(int a, int b) { return a > b ? a : b; }
int main()
{
    int x, z;
...
    z = imax(x, 50);
...
}
```

9.3　递归

C 允许函数调用它自己，这种调用过程称为*递归*（*recursion*）。递归有时难以捉摸，有时却很方便实用。结束递归是使用递归的难点，因为如果递归代码中没有终止递归的条件测试部分，一个调用自己的函数会无限递归。

可以使用循环的地方通常都可以使用递归。有时用循环解决问题比较好，但有时用递归更好。递归方案更简洁，但效率却没有循环高。

9.3.1　演示递归

我们通过一个程序示例，来学习什么是递归。程序清单 9.6 中的 main()函数调用 up_and_down()函数，这次调用称为“第 1 级递归”。然后 up_and_down()调用自己，这次调用称为“第 2 级递归”。接着第 2 级递归调用第 3 级递归，以此类推。该程序示例共有 4 级递归。为了进一步深入研究递归时发生了什么，程序不仅显示了变量 n 的值，还显示了储存 n 的内存地址&n。（本章稍后会详细讨论&运算符，printf()函数使用%p 转换说明打印地址，如果你的系统不支持这种格式，请使用%u 或%lu 代替%p）。

程序清单 9.6 recur.c 程序

```
/* recur.c -- 递归演示 */
#include <stdio.h>
void up_and_down(int);

int main(void)
{
    up_and_down(1);
    return 0;
}

void up_and_down(int n)
{
    printf("Level %d: n location %p\n", n, &n); // #1
    if (n < 4)
        up_and_down(n + 1);
    printf("LEVEL %d: n location %p\n", n, &n); // #2

}
```

下面是在我们系统中的输出：

```
Level 1: n location 0x0012ff48
Level 2: n location 0x0012ff3c
Level 3: n location 0x0012ff30
Level 4: n location 0x0012ff24
LEVEL 4: n location 0x0012ff24
LEVEL 3: n location 0x0012ff30
LEVEL 2: n location 0x0012ff3c
LEVEL 1: n location 0x0012ff48
```

我们来仔细分析程序中的递归是如何工作的。首先，main()调用了带参数 1 的 up_and_down()函数，执行结果是 up_and_down()中的形式参数 n 的值是 1，所以打印语句#1 打印 Level 1。然后，由于 n 小于 4，up_and_down()（第 1 级）调用实际参数为 n + 1（或 2）的 up_and_down()（第 2 级）。于是第 2 级调用中的 n 的值是 2，打印语句#1 打印 Level 2。与此类似，下面两次调用打印的分别是 Level 3 和 Level 4。

当执行到第 4 级时，n 的值是 4，所以 if 测试条件为假。up_and_down()函数不再调用自己。第 4 级调用接着执行打印语句#2，即打印 LEVEL 4，因为 n 的值是 4。此时，第 4 级调用结束，控制被传回它的主调函数（即第 3 级调用）。在第 3 级调用中，执行的最后一条语句是调用 if 语句中的第 4 级调用。被调函数（第 4 级调用）把控制返回在这个位置，因此，第 3 级调用继续执行后面的代码，打印语句#2 打印 LEVEL 3。然后第 3 级调用结束，控制被传回第 2 级调用，接着打印 LEVEL 2，以此类推。

注意，每级递归的变量 n 都属于本级递归私有。这从程序输出的地址值可以看出（当然，不同的系统表示的地址格式不同，这里关键要注意，Level 1 和 LEVEL 1 的地址相同，Level 2 和 LEVEL 2 的地址相同，等等）。

如果觉得不好理解，可以假设有一条函数调用链——fun1()调用 fun2()、fun2()调用 fun3()、fun3()调用 fun4()。当 fun4()结束时，控制传回 fun3()；当 fun3()结束时，控制传回 fun2()；当 fun2()结束时，控制传回 fun1()。递归的情况与此类似，只不过 fun1()、fun2()、fun3()和 fun4()都是相同的函数。

9.3.2　递归的基本原理

初次接触递归会觉得较难理解。为了帮助读者理解递归过程，下面以程序清单 9.6 为例讲解几个要点。

第 1，每级函数调用都有自己的变量。也就是说，第 1 级的 n 和第 2 级的 n 不同，所以程序创建了 4 个单独的变量，每个变量名都是 n，但是它们的值各不相同。当程序最终返回 up_and_down()的第 1 级调用时，最初的 n 仍然是它的初值 1（见图 9.4）。

变量	n	n	n	n
第1级调用后	1			
第2级调用后	1	2		
第3级调用后	1	2	3	
第4级调用后	1	2	3	4
从第4级调用返回后	1	2	3	
从第3级调用返回后	1	2		
从第2级调用返回后	1			
从第1级调用返回后	（全部结束）			

图 9.4　递归中的变量

第 2，每次函数调用都会返回一次。当函数执行完毕后，控制权将被传回上一级递归。程序必须按顺序逐级返回递归，从某级 up_and_down()返回上一级的 up_and_down()，不能跳级回到 main()中的第 1 级调用。

第 3，递归函数中位于递归调用之前的语句，均按被调函数的顺序执行。例如，程序清单 9.6 中的打印语句#1 位于递归调用之前，它按照递归的顺序：第 1 级、第 2 级、第 3 级和第 4 级，被执行了 4 次。

第 4，递归函数中位于递归调用之后的语句，均按被调函数相反的顺序执行。例如，打印语句#2 位于递归调用之后，其执行的顺序是第 4 级、第 3 级、第 2 级、第 1 级。递归调用的这种特性在解决涉及相反顺序的编程问题时很有用。稍后将介绍一个这样的例子。

第 5，虽然每级递归都有自己的变量，但是并没有拷贝函数的代码。程序按顺序执行函数中的代码，而递归调用就相当于又从头开始执行函数的代码。除了为每次递归调用创建变量外，递归调用非常类似于一个循环语句。实际上，递归有时可用循环来代替，循环有时也能用递归来代替。

最后，递归函数必须包含能让递归调用停止的语句。通常，递归函数都使用 if 或其他等价的测试条件在函数形参等于某特定值时终止递归。为此，每次递归调用的形参都要使用不同的值。例如，程序清单 9.6 中的 up_and_down(n)调用 up_and_down(n+1)。最终，实际参数等于 4 时，if 的测试条件(n < 4)为假。

9.3.3　尾递归

最简单的递归形式是把递归调用置于函数的末尾，即正好在 return 语句之前。这种形式的递归被称为*尾递归*（*tail recursion*），因为递归调用在函数的末尾。尾递归是最简单的递归形式，因为它相当于循环。

下面要介绍的程序示例中，分别用循环和尾递归计算阶乘。一个正整数的阶乘（*factorial*）是从 1 到该整数的所有整数的乘积。例如，3 的阶乘（写作 3!）是 1×2×3。另外，0! 等于 1，负数没有阶乘。程序清单 9.7 中，第 1 个函数使用 for 循环计算阶乘，第 2 个函数使用递归计算阶乘。

程序清单 9.7 factor.c **程序**

```
// factor.c -- 使用循环和递归计算阶乘
#include <stdio.h>
long fact(int n);
long rfact(int n);
int main(void)
{
    int num;

    printf("This program calculates factorials.\n");
    printf("Enter a value in the range 0-12 (q to quit):\n");
    while (scanf("%d", &num) == 1)
    {
        if (num < 0)
            printf("No negative numbers, please.\n");
        else if (num > 12)
            printf("Keep input under 13.\n");
        else
        {
            printf("loop: %d factorial = %ld\n",
                    num, fact(num));
            printf("recursion: %d factorial = %ld\n",
                    num, rfact(num));
        }
        printf("Enter a value in the range 0-12 (q to quit):\n");
    }
    printf("Bye.\n");

    return 0;
}

long fact(int n)     // 使用循环的函数
{
    long ans;

    for (ans = 1; n > 1; n--)
        ans *= n;

    return ans;
}

long rfact(int n)    // 使用递归的函数
{
    long ans;

    if (n > 0)
        ans = n * rfact(n - 1);
    else
        ans = 1;

    return ans;
}
```

测试驱动程序把输入限制在 0~12。因为 12!已快接近 5 亿，而 13!比 62 亿还大，已超过我们系统中

long 类型能表示的范围。要计算超过 12 的阶乘，必须使用能表示更大范围的类型，如 double 或 long long。

下面是该程序的运行示例：

```
This program calculates factorials.
Enter a value in the range 0-12 (q to quit):
5
loop: 5 factorial = 120
recursion: 5 factorial = 120
Enter a value in the range 0-12 (q to quit):
10
loop: 10 factorial = 3628800
recursion: 10 factorial = 3628800
Enter a value in the range 0-12 (q to quit):
q
Bye.
```

使用循环的函数把 ans 初始化为 1，然后把 ans 与从 n~2 的所有递减整数相乘。根据阶乘的公式，还应该乘以 1，但是这并不会改变结果。

现在考虑使用递归的函数。该函数的关键是 n! = n×(n-1)!。可以这样做是因为(n-1)!是 n-1~1 的所有正整数的乘积。因此，n 乘以 n-1 就得到 n 的阶乘。阶乘的这一特性很适合使用递归。如果调用函数 rfact()，rfact(n)是 n*rfact(n-1)。因此，通过调用 rfact(n-1)来计算 rfact(n)，如程序清单 9.7 中所示。当然，必须要在满足某条件时结束递归，可以在 n 等于 0 时把返回值设为 1。

程序清单 9.7 中使用递归的输出和使用循环的输出相同。注意，虽然 rfact()的递归调用不是函数的最后一行，但是当 n>0 时，它是该函数执行的最后一条语句，因此它也是尾递归。

既然用递归和循环来计算都没问题，那么到底应该使用哪一个？一般而言，选择循环比较好。首先，每次递归都会创建一组变量，所以递归使用的内存更多，而且每次递归调用都会把创建的一组新变量放在栈中。递归调用的数量受限于内存空间。其次，由于每次函数调用要花费一定的时间，所以递归的执行速度较慢。那么，演示这个程序示例的目的是什么？因为尾递归是递归中最简单的形式，比较容易理解。在某些情况下，不能用简单的循环代替递归，因此读者还是要好好理解递归。

9.3.4　递归和倒序计算

递归在处理倒序时非常方便（在解决这类问题中，递归比循环简单）。我们要解决的问题是：编写一个函数，打印一个整数的二进制数。二进制表示法根据 2 的幂来表示数字。例如，十进制数 234 实际上是 $2\times10^2+3\times10^1+4\times10^0$，所以二进制数 101 实际上是 $1\times2^2+0\times2^1+1\times2^0$。二进制数由 0 和 1 表示。

我们要设计一个以二进制形式表示整数的方法或*算法*（*algorithm*）。例如，如何用二进制表示十进制数 5？在二进制中，奇数的末尾一定是 1，偶数的末尾一定是 0，所以通过 5 % 2 即可确定 5 的二进制数的最后一位是 1 还是 0。一般而言，对于数字 n，其二进制的最后一位是 n % 2。因此，计算的第一位数字实际上是待输出二进制数的最后一位。这一规律提示我们，在递归函数的递归调用之前计算 n % 2，在递归调用之后打印计算结果。这样，计算的第 1 个值正好是最后一个打印的值。

要获得下一位数字，必须把原数除以 2。这种计算方法相当于在十进制下把小数点左移一位，如果计算结果是偶数，那么二进制的下一位数就是 0；如果是奇数，就是 1。例如，5/2 得 2（整数除法），2 是偶数（2%2 得 0），所以下一位二进制数是 0。到目前为止，我们已经获得 01。继续重复这个过程。2/2 得 1，1%2 得 1，所以下一位二进制数是 1。因此，我们得到 5 的等价二进制数是 101。那么，程序应该何时停止计算？当与 2 相除的结果小于 2 时停止计算，因为只要结果大于或等于 2，就说明还有二进制位。

每次除以 2 就相当于去掉一位二进制，直到计算出最后一位为止（如果不好理解，可以拿十进制数来做类比：628%10 得 8，因此 8 就是该数最后一位；而 628/10 得 62，而 62%10 得 2，所以该数的下一位是 2，以此类推）。程序清单 9.8 演示了上述算法。

程序清单 9.8 binary.c **程序**

```
/* binary.c -- 以二进制形式打印制整数 */
#include <stdio.h>
void to_binary(unsigned long n);

int main(void)
{
    unsigned long number;
    printf("Enter an integer (q to quit):\n");
    while (scanf("%lu", &number) == 1)
    {
        printf("Binary equivalent: ");
        to_binary(number);
        putchar('\n');
        printf("Enter an integer (q to quit):\n");
    }
    printf("Done.\n");

    return 0;
}

void to_binary(unsigned long n)   /* 递归函数 */
{
    int r;

    r = n % 2;
    if (n >= 2)
        to_binary(n / 2);
    putchar(r == 0 ? '0' : '1');

    return;
}
```

在该程序中，如果 r 的值是 0，to_binary()函数就显示字符'0'；如果 r 的值是 1，to_binary()函数则显示字符'1'。条件表达式 r == 0 ? '0' : '1'用于把数值转换成字符。

下面是该程序的运行示例：

```
Enter an integer (q to quit):
9
Binary equivalent: 1001
Enter an integer (q to quit):
255
Binary equivalent: 11111111
Enter an integer (q to quit):
1024
Binary equivalent: 10000000000
Enter an integer (q to quit):
q
done.
```

不用递归，是否能实现这种用二进制形式表示整数的算法？当然可以。但是由于这种算法要首先计算最后一位二进制数，所以在显示结果之前必须把所有的位数都储存在别处（例如，数组）。第 15 章中会介绍一个不用递归实现该算法的例子。

9.3.5　递归的优缺点

递归既有优点也有缺点。优点是递归为某些编程问题提供了最简单的解决方案。缺点是一些递归算法会快速消耗计算机的内存资源。另外，递归不方便阅读和维护。我们用一个例子来说明递归的优缺点。

斐波那契数列的定义如下：第 1 个和第 2 个数字都是 1，而后续的每个数字都是其前两个数字之和。例如，该数列的前几个数是：1、1、2、3、5、8、13。斐波那契数列在数学界深受喜爱，甚至有专门研究它的刊物。不过，这不在本书的讨论范围之内。下面，我们要创建一个函数，接受正整数 n，返回相应的斐波那契数值。

首先，来看递归。递归提供一个简单的定义。如果把函数命名为 Fibonacci()，那么如果 n 是 1 或 2，Fibonacci(n)应返回 1；对于其他数值，则应返回 Fibonacci(n-1)+Fibonacci(n-2)：

```
unsigned long Fibonacci(unsigned n)
{
    if (n > 2)
        return Fibonacci(n-1) + Fibonacci(n-2);
    else
        return 1;
}
```

这个递归函数只是重述了数学定义的递归。该函数使用了*双递归*（*double recursion*），即函数每一级递归都要调用本身两次。这暴露了一个问题。

为了说明这个问题，假设调用 Fibonacci(40)。这是第 1 级递归调用，将创建一个变量 n。然后在该函数中要调用 Fibonacci()两次，在第 2 级递归中要分别创建两个变量 n。这两次调用中的每次调用又会进行两次调用，因而在第 3 级递归中要创建 4 个名为 n 的变量。此时总共创建了 7 个变量。由于每级递归创建的变量都是上一级递归的两倍，所以变量的数量呈指数增长！在第 5 章中介绍过一个计算小麦粒数的例子，按指数增长很快就会产生非常大的值。在本例中，指数增长的变量数量很快就消耗掉计算机的大量内存，很可能导致程序崩溃。

虽然这是个极端的例子，但是该例说明：在程序中使用递归要特别注意，尤其是效率优先的程序。

所有的 C 函数皆平等

程序中的每个 C 函数与其他函数都是平等的。每个函数都可以调用其他函数，或被其他函数调用。这点与 Pascal 和 Modula-2 中的过程不同，虽然过程可以嵌套在另一个过程中，但是嵌套在不同过程中的过程之间不能相互调用。

main()函数是否与其他函数不同？是的，main()的确有点特殊。当 main()与程序中的其他函数放在一起时，最开始执行的是 main()函数中的第 1 条语句，但是这也是局限之处。main()也可以被自己或其他函数递归调用——尽管很少这样做。

9.4　编译多源代码文件的程序

使用多个函数最简单的方法是把它们都放在同一个文件中，然后像编译只有一个函数的文件那样编译

该文件即可。其他方法因操作系统而异，下面将举例说明。

9.4.1 UNIX

假定在 UNIX 系统中安装了 UNIX C 编译器 `cc`（最初的 `cc` 已经停用，但是许多 UNIX 系统都给 `cc` 命令起了一个别名用作其他编译器命令，典型的是 `gcc` 或 `clang`）。假设 `file1.c` 和 `file2.c` 是两个内含 C 函数的文件，下面的命令将编译两个文件并生成一个名为 `a.out` 的可执行文件：

```
cc file1.c file2.c
```

另外，还生成两个名为 `file1.o` 和 `file2.o` 的目标文件。如果后来改动了 `file1.c`，而 `file2.c` 不变，可以使用以下命令编译第 1 个文件，并与第 2 个文件的目标代码合并：

```
cc file1.c file2.o
```

UNIX 系统的 `make` 命令可自动管理多文件程序，但是这超出了本书的讨论范围。

注意，OS X 的 Terminal 工具可以打开 UNIX 命令行环境，但是必须先下载命令行编译器（GCC 和 Clang）。

9.4.2 Linux

假定 Linux 系统安装了 GNU C 编译器 GCC。假设 `file1.c` 和 `file2.c` 是两个内含 C 函数的文件，下面的命令将编译两个文件并生成名为 `a.out` 的可执行文件：

```
gcc file1.c file2.c
```

另外，还生成两个名为 `file1.o` 和 `file2.o` 的目标文件。如果后来改动了 `file1.c`，而 `file2.c` 不变，可以使用以下命令编译第 1 个文件，并与第 2 个文件的目标代码合并：

```
gcc file1.c file2.o
```

9.4.3 DOS 命令行编译器

绝大多数 DOS 命令行编译器的工作原理和 UNIX 的 `cc` 命令类似，只不过使用不同的名称而已。其中一个区别是，对象文件的扩展名是`.obj`，而不是`.o`。一些编译器生成的不是目标代码文件，而是汇编语言或其他特殊代码的中间文件。

9.4.4 Windows 和苹果的 IDE 编译器

Windows 和 Macintosh 系统使用的集成开发环境中的编译器是面向项目的。项目（*project*）描述的是特定程序使用的资源。资源包括源代码文件。这种 IDE 中的编译器要创建项目来运行单文件程序。对于多文件程序，要使用相应的菜单命令，把源代码文件加入一个项目中。要确保所有的源代码文件都在项目列表中列出。许多 IDE 都不用在项目列表中列出头文件（即扩展名为`.h` 的文件），因为项目只管理使用的源代码文件，源代码文件中的`#include` 指令管理该文件中使用的头文件。但是，Xcode 要在项目中添加头文件。

9.4.5 使用头文件

如果把 `main()`放在第 1 个文件中，把函数定义放在第 2 个文件中，那么第 1 个文件仍然要使用函数原型。把函数原型放在头文件中，就不用在每次使用函数文件时都写出函数的原型。C 标准库就是这样做的，例如，把 I/O 函数原型放在 `stdio.h` 中，把数学函数原型放在 `math.h` 中。你也可以这样用自定义的函数文件。

另外，程序中经常用 C 预处理器定义符号常量。这种定义只储存了那些包含`#define` 指令的文件。如果把程序的一个函数放进一个独立的文件中，你也可以使用`#define` 指令访问每个文件。最直接的方法是

在每个文件中再次输入指令，但是这个方法既耗时又容易出错。另外，还会有维护的问题：如果修改了#define 定义的值，就必须在每个文件中修改。更好的做法是，把#define 指令放进头文件，然后在每个源文件中使用#include 指令包含该文件即可。

总之，把函数原型和已定义的字符常量放在头文件中是一个良好的编程习惯。我们考虑一个例子：假设要管理 4 家酒店的客房服务，每家酒店的房价不同，但是每家酒店所有房间的房价相同。对于预订住宿多天的客户，第 2 天的房费是第 1 天的 95%，第 3 天是第 2 天的 95%，以此类推（暂不考虑这种策略的经济效益）。设计一个程序让用户指定酒店和入住天数，然后计算并显示总费用。同时，程序要实现一份菜单，允许用户反复输入数据，除非用户选择退出。

程序清单 9.9、程序清单 9.10 和程序清单 9.11 演示了如何编写这样的程序。第 1 个程序清单包含 main()函数，提供整个程序的组织结构。第 2 个程序清单包含支持的函数，我们假设这些函数在独立的文件中。最后，程序清单 9.11 列出了一个头文件，包含了该程序所有源文件中使用的自定义符号常量和函数原型。前面介绍过，在 UNIX 和 DOS 环境中，#include "hotels.h"指令中的双引号表明被包含的文件位于当前目录中（通常是包含源代码的目录）。如果使用 IDE，需要知道如何把头文件合并成一个项目。

程序清单 9.9　usehotel.c 控制模块

```
/* usehotel.c -- 房间费率程序 */
/* 与程序清单 9.10 一起编译      */
#include <stdio.h>
#include "hotel.h" /* 定义符号常量，声明函数 */

int main(void)
{
    int nights;
    double hotel_rate;
    int code;

    while ((code = menu()) != QUIT)
    {
        switch (code)
        {
            case 1:  hotel_rate = HOTEL1;
                     break;
            case 2:  hotel_rate = HOTEL2;
                     break;
            case 3:  hotel_rate = HOTEL3;
                     break;
            case 4:  hotel_rate = HOTEL4;
                     break;
            default: hotel_rate = 0.0;
                     printf("Oops!\n");
                     break;
        }
        nights = getnights();
        showprice(hotel_rate, nights);
    }
    printf("Thank you and goodbye.\n");

    return 0;
}
```

程序清单 9.10 `hotel.c` **函数支持模块**

```
/* hotel.c -- 酒店管理函数 */
#include <stdio.h>
#include "hotel.h"
int menu(void)
{
    int code, status;

    printf("\n%s%s\n", STARS, STARS);
    printf("Enter the number of the desired hotel:\n");
    printf("1) Fairfield Arms          2) Hotel Olympic\n");
    printf("3) Chertworthy Plaza       4) The Stockton\n");
    printf("5) quit\n");
    printf("%s%s\n", STARS, STARS);
    while ((status = scanf("%d", &code)) != 1 ||
            (code < 1 || code > 5))
    {
        if (status != 1)
            scanf("%*s");   // 处理非整数输入
        printf("Enter an integer from 1 to 5, please.\n");
    }

    return code;
}

int getnights(void)
{
    int nights;

    printf("How many nights are needed? ");
    while (scanf("%d", &nights) != 1)
    {
        scanf("%*s");      // 处理非整数输入
        printf("Please enter an integer, such as 2.\n");
    }

    return nights;
}

void showprice(double rate, int nights)
{
    int n;
    double total = 0.0;
    double factor = 1.0;

    for (n = 1; n <= nights; n++, factor *= DISCOUNT)
        total += rate * factor;
    printf("The total cost will be $%0.2f.\n", total);
}
```

程序清单 9.11 `hotel.h` **头文件**

```
/* hotel.h -- 符号常量和 hotel.c 中所有函数的原型 */
#define QUIT      5
```

```
#define HOTEL1   180.00
#define HOTEL2   225.00
#define HOTEL3   255.00
#define HOTEL4   355.00
#define DISCOUNT  0.95
#define STARS "**********************************"

// 显示选择列表
int menu(void);

// 返回预订天数
int getnights(void);

// 根据费率、入住天数计算费用
// 并显示结果
void showprice(double rate, int nights);
```

下面是这个多文件程序的运行示例：

```
*****************************************************************
Enter the number of the desired hotel:
1) Fairfield Arms          2) Hotel Olympic
3) Chertworthy Plaza       4) The Stockton
5) quit
*****************************************************************
3
How many nights are needed? 1
The total cost will be $255.00.

*****************************************************************
Enter the number of the desired hotel:
1) Fairfield Arms          2) Hotel Olympic
3) Chertworthy Plaza       4) The Stockton
5) quit
*****************************************************************
4
How many nights are needed? 3
The total cost will be $1012.64.

*****************************************************************
Enter the number of the desired hotel:
1) Fairfield Arms          2) Hotel Olympic
3) Chertworthy Plaza       4) The Stockton
5) quit
*****************************************************************
5
Thank you and goodbye.
```

顺带一提，该程序中有几处编写得很巧妙。尤其是，menu()和 getnights()函数通过测试 scanf()的返回值来跳过非数值数据，而且调用 scanf("%*s")跳至下一个空白字符。注意，menu()函数中是如何检查非数值输入和超出范围的数据：

```
while ((status = scanf("%d", &code)) != 1 ||(code < 1 || code > 5))
```

以上代码段利用了 C 语言的两个规则：从左往右对逻辑表达式求值；一旦求值结果为假，立即停止求值。在该例中，只有在 scanf()成功读入一个整数值后，才会检查 code 的值。

用不同的函数处理不同的任务时应检查数据的有效性。当然，首次编写 menu()或 getnights()函数时可以暂不添加这一功能，只写一个简单的 scanf()即可。待基本版本运行正常后，再逐步改善各模块。

9.5　查找地址：&运算符

指针（*pointer*）是 C 语言最重要的（有时也是最复杂的）概念之一，用于储存变量的地址。前面使用的 scanf()函数中就使用地址作为参数。概括地说，如果主调函数不使用 return 返回的值，则必须通过地址才能修改主调函数中的值。接下来，我们将介绍带地址参数的函数。首先介绍一元&运算符的用法。

一元&运算符给出变量的存储地址。如果 pooh 是变量名，那么&pooh 是变量的地址。可以把地址看作是变量在内存中的位置。假设有下面的语句：

```
pooh = 24;
```

假设 pooh 的存储地址是 0B76（PC 地址通常用十六进制形式表示）。那么，下面的语句：

```
printf("%d %p\n", pooh, &pooh);
```

将输出如下内容（%p 是输出地址的转换说明）：

```
24 0B76
```

程序清单 9.12 中使用了这个运算符查看不同函数中的同名变量分别储存在什么位置。

程序清单 9.12　loccheck.c 程序

```
/* loccheck.c  -- 查看变量被储存在何处  */
#include <stdio.h>
void mikado(int);               /* 函数原型 */
int main(void)
{
    int pooh = 2, bah = 5; /* main()的局部变量 */

    printf("In main(), pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("In main(), bah = %d and &bah = %p\n", bah, &bah);
    mikado(pooh);

    return 0;
}

void mikado(int bah)            /* 定义函数 */
{
    int pooh = 10;              /* mikado()的局部变量 */

    printf("In mikado(), pooh = %d and &pooh = %p\n", pooh, &pooh);
    printf("In mikado(), bah = %d and &bah = %p\n",  bah, &bah);
}
```

程序清单 9.12 中使用 ANSI C 的%p 格式打印地址。我们的系统输出如下：

```
In main(), pooh = 2 and &pooh = 0x7fff5fbff8e8
In main(), bah = 5 and &bah = 0x7fff5fbff8e4
In mikado(), pooh = 10 and &pooh = 0x7fff5fbff8b8
In mikado(), bah = 2 and &bah = 0x7fff5fbff8bc
```

实现不同，%p 表示地址的方式也不同。然而，许多实现都如本例所示，以十六进制显示地址。顺带一提，每个十六进制数对应 4 位，该例显示 12 个十六进制数，对应 48 位地址。

该例的输出说明了什么？首先，两个 pooh 的地址不同，两个 bah 的地址也不同。因此，和前面介绍的一样，计算机把它们看成 4 个独立的变量。其次，函数调用 mikado(pooh)把实际参数（main()中的 pooh）的值（2）传递给形式参数（mikado()中的 bah）。注意，这种传递只传递了值。涉及的两个变量（main()中的 pooh 和 mikado()中的 bah）并未改变。

我们强调第 2 点，是因为这并不是在所有语言中都成立。例如，在 FORTRAN 中，子例程会影响主调例程的原始变量。子例程的变量名可能与原始变量不同，但是它们的地址相同。但是，在 C 语言中不是这样。每个 C 函数都有自己的变量。这样做更可取，因为这样做可以防止原始变量被被调函数中的副作用意外修改。然而，正如下节所述，这也带来了一些麻烦。

9.6　更改主调函数中的变量

有时需要在一个函数中更改其他函数的变量。例如，普通的排序任务中交换两个变量的值。假设要交换两个变量 x 和 y 的值。简单的思路是：

```
x = y;
y = x;
```

这完全不起作用，因为执行到第 2 行时，x 的原始值已经被 y 的原始值替换了。因此，要多写一行代码，储存 x 的原始值：

```
temp = x;
x = y;
y = temp;
```

上面这 3 行代码便可实现交换值的功能，可以编写成一个函数并构造一个驱动程序来测试。在程序清单 9.13 中，为清楚地表明变量属于哪个函数，在 main()中使用变量 x 和 y，在 intercharge()中使用 u 和 v。

程序清单 9.13　swap1.c 程序

```
/* swap1.c -- 第 1 个版本的交换函数 */
#include <stdio.h>
void interchange(int u, int v); /* 声明函数 */

int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)  /* 定义函数  */
{
    int temp;

    temp = u;
    u = v;
    v = temp;
}
```

运行该程序后，输出如下：

```
Originally x = 5 and y = 10.
Now x = 5 and y = 10.
```

两个变量的值并未交换！我们在 `interchange()` 中添加一些打印语句来检查错误（见程序清单 9.14）。

程序清单 9.14 swap2.c 程序

```
/* swap2.c -- 查找 swap1.c 的问题 */
#include <stdio.h>
void interchange(int u, int v);

int main(void)
{
    int x = 5, y = 10;

    printf("Originally x = %d and y = %d.\n", x, y);
    interchange(x, y);
    printf("Now x = %d and y = %d.\n", x, y);

    return 0;
}

void interchange(int u, int v)
{
    int temp;

    printf("Originally u = %d and v = %d.\n", u, v);
    temp = u;
    u = v;
    v = temp;
    printf("Now u = %d and v = %d.\n", u, v);
}
```

下面是该程序的输出：

```
Originally x = 5 and y = 10.
Originally u = 5 and v = 10.
Now u = 10 and v = 5.
Now x = 5 and y = 10.
```

看来，`interchange()` 没有问题，它交换了 `u` 和 `v` 的值。问题出在把结果传回 `main()` 时。`interchange()` 使用的变量并不是 `main()` 中的变量。因此，交换 `u` 和 `v` 的值对 `x` 和 `y` 的值没有影响！是否能用 `return` 语句把值传回 `main()`？当然可以，在 `interchange()` 的末尾加上下面一行语句：

```
return(u);
```

然后修改 `main()` 中的调用：

```
x = interchange(x,y);
```

这只能改变 `x` 的值，而 `y` 的值依旧没变。用 `return` 语句只能把被调函数中的一个值传回主调函数，但是现在要传回两个值。这没问题！不过，要使用指针。

9.7 指针简介

指针？什么是指针？从根本上看，指针（*pointer*）是一个值为内存地址的变量（或数据对象）。正如 `char` 类型变量的值是字符，`int` 类型变量的值是整数，指针变量的值是地址。在 C 语言中，指针有许多用法。

本章将介绍如何把指针作为函数参数使用，以及为何要这样用。

假设一个指针变量名是 ptr，可以编写如下语句：

```
ptr = &pooh;  // 把 pooh 的地址赋给 ptr
```

对于这条语句，我们说 ptr“指向”pooh。ptr 和&pooh 的区别是 ptr 是变量，而&pooh 是常量。或者，ptr 是可修改的左值，而&pooh 是右值。还可以把 ptr 指向别处：

```
ptr = &bah;    // 把 ptr 指向 bah，而不是 pooh
```

现在 ptr 的值是 bah 的地址。

要创建指针变量，先要声明指针变量的类型。假设想把 ptr 声明为储存 int 类型变量地址的指针，就要使用下面介绍的新运算符。

9.7.1　间接运算符：*

假设已知 ptr 指向 bah，如下所示：

```
ptr = &bah;
```

然后使用间接运算符*（*indirection operator*）找出储存在 bah 中的值，该运算符有时也称为解引用运算符（*dereferencing operator*）。不要把间接运算符和二元乘法运算符（*）混淆，虽然它们使用的符号相同，但语法功能不同。

```
val = *ptr; // 找出 ptr 指向的值
```

语句 ptr = &bah;和 val = *ptr;放在一起相当于下面的语句：

```
val = bah;
```

由此可见，使用地址和间接运算符可以间接完成上面这条语句的功能，这也是“间接运算符”名称的由来。

> **小结：与指针相关的运算符**
>
> **地址运算符：&**
>
> **一般注解：**
>
> 后跟一个变量名时，&给出该变量的地址。
>
> **示例：**
>
> &nurse 表示变量 nurse 的地址。
>
> **地址运算符：***
>
> **一般注解：**
>
> 后跟一个指针名或地址时，*给出储存在指针指向地址上的值。
>
> **示例：**
>
> ```
> nurse = 22;
> ptr = &nurse; // 指向 nurse 的指针
> val = *ptr;   // 把 ptr 指向的地址上的值赋给 val
> ```
>
> 执行以上 3 条语句的最终结果是把 22 赋给 val。

9.7.2　声明指针

相信读者已经很熟悉如何声明 int 类型和其他基本类型的变量，那么如何声明指针变量？你也许认为

是这样声明：

```
pointer ptr; // 不能这样声明指针
```

为什么不能这样声明？因为声明指针变量时必须指定指针所指向变量的类型，因为不同的变量类型占用不同的存储空间，一些指针操作要求知道操作对象的大小。另外，程序必须知道储存在指定地址上的数据类型。long 和 float 可能占用相同的存储空间，但是它们储存数字却大相径庭。下面是一些指针的声明示例：

```
int * pi;      // pi 是指向 int 类型变量的指针
char * pc;          // pc 是指向 char 类型变量的指针
float * pf, * pg;  // pf、pg 都是指向 float 类型变量的指针
```

类型说明符表明了指针所指向对象的类型，星号（*）表明声明的变量是一个指针。int * pi;声明的意思是 pi 是一个指针，*pi 是 int 类型（见图 9.5）。

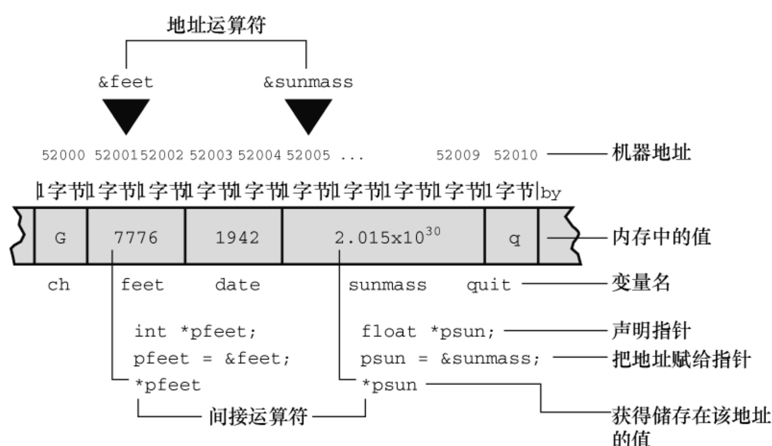


图 9.5　声明并使用指针

*和指针名之间的空格可有可无。通常，程序员在声明时使用空格，在解引用变量时省略空格。

pc 指向的值（*pc）是 char 类型。pc 本身是什么类型？我们描述它的类型是“指向 char 类型的指针”。pc 的值是一个地址，在大部分系统内部，该地址由一个无符号整数表示。但是，不要把指针认为是整数类型。一些处理整数的操作不能用来处理指针，反之亦然。例如，可以把两个整数相乘，但是不能把两个指针相乘。所以，指针实际上是一个新类型，不是整数类型。因此，如前所述，ANSI C 专门为指针提供了%p 格式的转换说明。

9.7.3　使用指针在函数间通信

我们才刚刚接触指针，指针的世界丰富多彩。本节着重介绍如何使用指针解决函数间的通信问题。请看程序清单 9.15，该程序在 interchange()函数中使用了指针参数。稍后我们将对该程序做详细分析。

程序清单 9.15　swap3.c **程序**

```
/* swap3.c -- 使用指针解决交换函数的问题 */
#include <stdio.h>
void interchange(int * u, int * v);

int main(void)
{
    int x = 5, y = 10;
    printf("Originally x = %d and y = %d.\n", x, y);
```

```
        interchange(&x, &y);   // 把地址发送给函数
        printf("Now x = %d and y = %d.\n", x, y);

        return 0;
    }

    void interchange(int * u, int * v)
    {
        int temp;
        temp = *u;     // temp 获得 u 所指向对象的值
        *u = *v;
        *v = temp;
    }
```

该程序是否能正常运行？下面是程序的输出：

```
Originally x = 5 and y = 10.
Now x = 10 and y = 5.
```

没问题，一切正常。接下来，我们分析程序清单 9.15 的运行情况。首先看函数调用：

```
interchange(&x, &y);
```

该函数传递的不是 x 和 y 的值，而是它们的地址。这意味着出现在 interchange()原型和定义中的形式参数 u 和 v 将把地址作为它们的值。因此，应把它们声明为指针。由于 x 和 y 是整数，所以 u 和 v 是指向整数的指针，其声明如下：

```
void interchange (int * u, int * v)
```

接下来，在函数体中声明了一个交换值时必需的临时变量：

```
int temp;
```

通过下面的语句把 x 的值储存在 temp 中：

```
temp = *u;
```

记住，u 的值是&x，所以 u 指向 x。这意味着用*u 即可表示 x 的值，这正是我们需要的。不要写成这样：

```
temp = u; /* 不要这样做 */
```

因为这条语句赋给 temp 的是 x 的地址（u 的值就是 x 的地址），而不是 x 的值。函数要交换的是 x 和 y 的值，而不是它们的地址。

与此类似，把 y 的值赋给 x，要使用下面的语句：

```
*u = *v;
```

这条语句相当于：

```
x = y;
```

我们总结一下该程序示例做了什么。我们需要一个函数交换 x 和 y 的值。把 x 和 y 的地址传递给函数，我们让 interchange()访问这两个函数。使用指针和*运算符，该函数可以访问储存在这些位置的值并改变它们。

可以省略 ANSI C 风格的函数原型中的形参名，如下所示：

```
void interchange(int *, int *);
```

一般而言，可以把变量相关的两类信息传递给函数。如果这种形式的函数调用，那么传递的是 x 的值：

```
function1(x);
```

如果下面形式的函数调用，那么传递的是 x 的地址：

```
function2(&x);
```

第 1 种形式要求函数定义中的形式参数必须是一个与 x 的类型相同的变量：

```
int function1(int num)
```

第 2 种形式要求函数定义中的形式参数必须是一个指向正确类型的指针：

```
int function2(int * ptr)
```

如果要计算或处理值，那么使用第 1 种形式的函数调用；如果要在被调函数中改变主调函数的变量，则使用第 2 种形式的函数调用。我们用过的 scanf()函数就是这样。当程序要把一个值读入变量时（如本例中的 num），调用的是 scanf("%d", &num)。scanf()读取一个值，然后把该值储存到指定的地址上。

对本例而言，指针让 interchange()函数通过自己的局部变量改变 main()中变量的值。

熟悉 Pascal 和 Modula-2 的读者应该看出第 1 种形式和 Pascal 的值参数相同，第 2 种形式和 Pascal 的变量参数类似。C++程序员可能认为，既然 C 和 C++都使用指针变量，那么 C 应该也有引用变量。让他们失望了，C 没有引用变量。对 BASIC 程序员而言，可能很难理解整个程序。如果觉得本节的内容晦涩难懂，请多做一些相关的编程练习，你会发现指针非常简单实用（见图 9.6）。

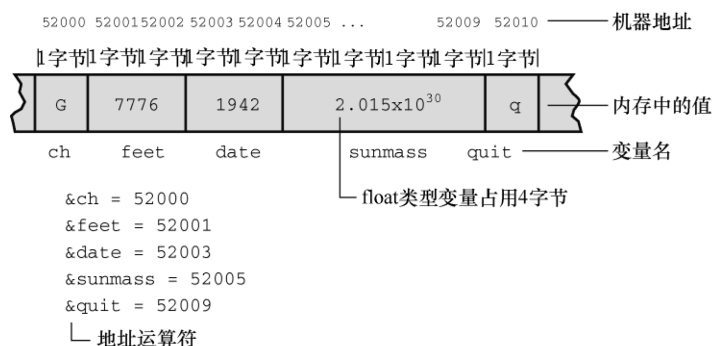


图 9.6 按字节寻址系统（如 PC）中变量的名称、地址和值

变量：名称、地址和值

通过前面的讨论发现，变量的名称、地址和变量的值之间关系密切。我们来进一步分析。

编写程序时，可以认为变量有两个属性：名称和值（还有其他性质，如类型，暂不讨论）。计算机编译和加载程序后，认为变量也有两个属性：地址和值。地址就是变量在计算机内部的名称。

在许多语言中，地址都归计算机管，对程序员隐藏。然而在 C 中，可以通过&运算符访问地址，通过*运算符获得地址上的值。例如，&barn 表示变量 barn 的地址，使用函数名即可获得变量的数值。例如，printf("%d\n", barn)打印 barn 的值，使用*运算符即可获得储存在地址上的值。如果 pbarn = &barn;，那么*pbarn 表示的是储存在&barn 地址上的值。

简而言之，普通变量把值作为基本量，把地址作为通过&运算符获得的派生量，而指针变量把地址作为基本量，把值作为通过*运算符获得的派生量。

虽然打印地址可以满足读者好奇心，但是这并不是&运算符的主要用途。更重要的是使用&、*和指针可以操纵地址和地址上的内容，如 swap3.c 程序（程序清单 9.15）所示。

小结：函数

形式：

典型的 ANSI C 函数的定义形式为：

返回类型 名称（形参声明列表）

函数体

形参声明列表是用逗号分隔的一系列变量声明。除形参变量外，函数的其他变量均在函数体的花括号之内声明。

示例：

```
int diff(int x, int y) // ANSI C
{ // 函数体开始
    int z;         // 声明局部变量
    z = x - y;
    return z; // 返回一个值
} // 函数体结束
```

传递值：

实参用于把值从主调函数传递给被调函数。如果变量 a 和 b 的值分别是 5 和 2，那么调用：

```
c = diff(a,b);
```

把 5 和 2 分别传递给变量 x 和 y。5 和 2 称为实际参数（简称实参），diff()函数定义中的变量 x 和 y 称为形式参数（简称形参）。使用关键字 return 把被调函数中的一个值传回主调函数。本例中，c 接受 z 的值 3。被调函数一般不会改变主调函数中的变量，如果要改变，应使用指针作为参数。如果希望把更多的值传回主调函数，必须这么做。

函数的返回类型：

函数的返回类型指的是函数返回值的类型。如果返回值的类型与声明的返回类型不匹配，返回值将被转换成函数声明的返回类型。

函数签名：

函数的返回类型和形参列表构成了函数签名。因此，函数签名指定了传入函数的值的类型和函数返回值的类型。

示例：

```
double duff(double, int);  // 函数原型
int main(void)
{
    double q, x;
    int n;
    ...
    q = duff(x,n);         //函数调用
    ...
}
double duff(double u, int k)   //函数定义
{
    double tor;
    ...
    return tor;   //返回 double 类型的值
}
```

9.8 关键概念

如果想用 C 编出高效灵活的程序，必须理解函数。把大型程序组织成若干函数非常有用，甚至很关键。如果让一个函数处理一个任务，程序会更好理解，更方便调试。要理解函数是如何把信息从一个函数传递

到另一函数，也就是说，要理解函数参数和返回值的工作原理。另外，要明白函数形参和其他局部变量都属于函数私有，因此，声明在不同函数中的同名变量是完全不同的变量。而且，函数无法直接访问其他函数中的变量。这种限制访问保护了数据的完整性。但是，当确实需要在函数中访问另一个函数的数据时，可以把指针作为函数的参数。

9.9 本章小结

函数可以作为组成大型程序的构件块。每个函数都应该有一个单独且定义好的功能。使用参数把值传给函数，使用关键字 `return` 把值返回函数。如果函数返回的值不是 `int` 类型，则必须在函数定义和函数原型中指定函数的类型。如果需要在被调函数中修改主调函数的变量，使用地址或指针作为参数。

ANSI C 提供了一个强大的工具——*函数原型*，允许编译器验证函数调用中使用的参数个数和类型是否正确。

C 函数可以调用本身，这种调用方式被称为*递归*。一些编程问题要用递归来解决，但是递归不仅消耗内存多，效率不高，而且费时。

9.10 复习题

复习题的参考答案在附录 A 中。

1. 实际参数和形式参数的区别是什么？
2. 根据下面各函数的描述，分别编写它们的 ANSI C 函数头。注意，只需写出函数头，不用写函数体。
   a. `donut()`接受一个 `int` 类型的参数，打印若干（参数指定数目）个 `0`
   b. `gear()`接受两个 `int` 类型的参数，返回 `int` 类型的值
   c. `guess()`不接受参数，返回一个 `int` 类型的值
   d. `stuff_it()`接受一个 `double` 类型的值和 `double` 类型变量的地址，把第 `1` 个值储存在指定位置
3. 根据下面各函数的描述，分别编写它们的 ANSI C 函数头。注意，只需写出函数头，不用写函数体。
   a. `n_to_char()`接受一个 `int` 类型的参数，返回一个 `char` 类型的值
   b. `digit()`接受一个 `double` 类型的参数和一个 `int` 类型的参数，返回一个 `int` 类型的值
   c. `which()`接受两个可储存 `double` 类型变量的地址，返回一个 `double` 类型的地址
   d. `random()`不接受参数，返回一个 `int` 类型的值
4. 设计一个函数，返回两整数之和。
5. 如果把复习题 4 改成返回两个 `double` 类型的值之和，应如何修改函数？
6. 设计一个名为 `alter()`的函数，接受两个 `int` 类型的变量 `x` 和 `y`，把它们的值分别改成两个变量之和以及两变量之差。
7. 下面的函数定义是否正确？

```
void salami(num)
{
     int num, count;
     for (count = 1; count <= num; num++)
         printf(" O salami mio!\n");
}
```

8. 编写一个函数，返回 3 个整数参数中的最大值。
9. 给定下面的输出：

```
Please choose one of the following:
1) copy files           2) move files
3) remove files         4) quit
Enter the number of your choice:
```

a．编写一个函数，显示一份有 4 个选项的菜单，提示用户进行选择（输出如上所示）。

b．编写一个函数，接受两个 `int` 类型的参数分别表示上限和下限。该函数从用户的输入中读取整数。如果整数超出规定上下限，函数再次打印菜单（使用 a 部分的函数）提示用户输入，然后获取一个新值。如果用户输入的整数在规定范围内，该函数则把该整数返回主调函数。如果用户输入一个非整数字符，该函数应返回 4。

c．使用本题 a 和 b 部分的函数编写一个最小型的程序。最小型的意思是，该程序不需要实现菜单中各选项的功能，只需显示这些选项并获取有效的响应即可。

9.11　编程练习

1. 设计一个函数 `min(x, y)`，返回两个 `double` 类型值的较小值。在一个简单的驱动程序中测试该函数。
2. 设计一个函数 `chline(ch, i, j)`，打印指定的字符 j 行 i 列。在一个简单的驱动程序中测试该函数。
3. 编写一个函数，接受 3 个参数：一个字符和两个整数。字符参数是待打印的字符，第 1 个整数指定一行中打印字符的次数，第 2 个整数指定打印指定字符的行数。编写一个调用该函数的程序。
4. 两数的调和平均数这样计算：先得到两数的倒数，然后计算两个倒数的平均值，最后取计算结果的倒数。编写一个函数，接受两个 `double` 类型的参数，返回这两个参数的调和平均数。
5. 编写并测试一个函数 `larger_of()`，该函数把两个 `double` 类型变量的值替换为较大的值。例如，`larger_of(x, y)`会把 x 和 y 中较大的值重新赋给两个变量。
6. 编写并测试一个函数，该函数以 3 个 `double` 变量的地址作为参数，把最小值放入第 1 个函数，中间值放入第 2 个变量，最大值放入第 3 个变量。
7. 编写一个函数，从标准输入中读取字符，直到遇到文件结尾。程序要报告每个字符是否是字母。如果是，还要报告该字母在字母表中的数值位置。例如，c 和 C 在字母表中的位置都是 3。合并一个函数，以一个字符作为参数，如果该字符是一个字母则返回一个数值位置，否则返回-1。
8. 第 6 章的程序清单 6.20 中，`power()`函数返回一个 `double` 类型数的正整数次幂。改进该函数，使其能正确计算负幂。另外，函数要处理 0 的任何次幂都为 0，任何数的 0 次幂都为 1（函数应报告 0 的 0 次幂未定义，因此把该值处理为 1）。要使用一个循环，并在程序中测试该函数。
9. 使用递归函数重写编程练习 8。
10. 为了让程序清单 9.8 中的 `to_binary()`函数更通用，编写一个 `to_base_n()`函数接受两个在 2～10 范围内的参数，然后以第 2 个参数中指定的进制打印第 1 个参数的数值。例如，`to_base_n(129, 8)`显示的结果为 201，也就是 129 的八进制数。在一个完整的程序中测试该函数。
11. 编写并测试 `Fibonacci()`函数，该函数用循环代替递归计算斐波那契数。

第10章

数组和指针

本章介绍以下内容：

- 关键字：static
- 运算符：&、*（一元）
- 如何创建并初始化数组
- 指针（在已学过的基础上）、指针和数组的关系
- 编写处理数组的函数
- 二维数组

人们通常借助计算机完成统计每月的支出、日降雨量、季度销售额等任务。企业借助计算机管理薪资、库存和客户交易记录等。作为程序员，不可避免地要处理大量相关数据。通常，数组能高效便捷地处理这种数据。第 6 章简单地介绍了数组，本章将进一步地学习如何使用数组，着重分析如何编写处理数组的函数。这种函数把模块化编程的优势应用到数组。通过本章的学习，你将明白数组和指针关系密切。

10.1 数组

前面介绍过，数组由数据类型相同的一系列元素组成。需要使用数组时，通过声明数组告诉编译器数组中内含多少元素和这些元素的类型。编译器根据这些信息正确地创建数组。普通变量可以使用的类型，数组元素都可以用。考虑下面的数组声明：

```
/* 一些数组声明*/
int main(void)
{
    float candy[365];      /* 内含 365 个 float 类型元素的数组 */
    char code[12];         /*内含 12 个 char 类型元素的数组*/
    int states[50];        /*内含 50 个 int 类型元素的数组 */
    ...
}
```

方括号（[]）表明 candy、code 和 states 都是数组，方括号中的数字表明数组中的元素个数。

要访问数组中的元素，通过使用数组下标数（也称为索引）表示数组中的各元素。数组元素的编号从 0 开始，所以 candy[0]表示 candy 数组的第 1 个元素，candy[364]表示第 365 个元素，也就是最后一个元素。读者对这些内容应该比较熟悉，下面我们介绍一些新内容。

10.1.1 初始化数组

数组通常被用来储存程序需要的数据。例如，一个内含 12 个整数元素的数组可以储存 12 个月的天数。在这种情况下，在程序一开始就初始化数组比较好。下面介绍初始化数组的方法。

只储存单个值的变量有时也称为标量变量（*scalar variable*），我们已经很熟悉如何初始化这种变量：

```
int fix = 1;
float flax = PI * 2;
```

代码中的 PI 已定义为宏。C 使用新的语法来初始化数组，如下所示：

```
int main(void)
{
    int powers[8] = {1,2,4,6,8,16,32,64}; /* 从 ANSI C 开始支持这种初始化 */
    ...
}
```

如上所示，用以逗号分隔的值列表（用花括号括起来）来初始化数组，各值之间用逗号分隔。在逗号和值之间可以使用空格。根据上面的初始化，把 1 赋给数组的首元素（powers[0]），以此类推（不支持 ANSI 的编译器会把这种形式的初始化识别为语法错误，在数组声明前加上关键字 static 可解决此问题。第 12 章将详细讨论这个关键字）。

程序清单 10.1 演示了一个小程序，打印每个月的天数。

程序清单 10.1　day_mon1.c 程序

```
/* day_mon1.c -- 打印每个月的天数 */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %2d has %2d days.\n", index + 1, days[index]);

    return 0;
}
```

该程序的输出如下：

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
Month 11 has 30 days.
Month 12 has 31 days.
```

这个程序还不够完善，每 4 年打错一个月份的天数（即，2 月份的天数）。该程序用初始化列表初始化 days[]，列表（用花括号括起来）中用逗号分隔各值。

注意该例使用了符号常量 MONTHS 表示数组大小，这是我们推荐且常用的做法。例如，如果要采用一年 13 个月的记法，只需修改#define 这行代码即可，不用在程序中查找所有使用过数组大小的地方。

注意 使用 const 声明数组

有时需要把数组设置为只读。这样，程序只能从数组中检索值，不能把新值写入数组。要创建只读数组，应该用 const 声明和初始化数组。因此，程序清单 10.1 中初始化数组应改成:

```
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

这样修改后，程序在运行过程中就不能修改该数组中的内容。和普通变量一样，应该使用声明来初始化 const 数据，因为一旦声明为 const，便不能再给它赋值。明确了这一点，就可以在后面的例子中使用 const 了。

如果初始化数组失败怎么办？程序清单 10.2 演示了这种情况。

程序清单 10.2 no_data.c 程序

```
/* no_data.c -- 为初始化数组 */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int no_data[SIZE];  /* 未初始化数组 */
    int i;

    printf("%2s%14s\n",    "i", "no_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, no_data[i]);

    return 0;
}
```

该程序的输出如下（系统不同，输出的结果可能不同）:

```
i    no_data[i]
0             0
1       4204937
2       4219854
3    2147348480
```

使用数组前必须先初始化它。与普通变量类似，在使用数组元素之前，必须先给它们赋初值。编译器使用的值是内存相应位置上的现有值，因此，读者运行该程序后的输出会与该示例不同。

注意 存储类别警告

数组和其他变量类似，可以把数组创建成不同的存储类别（*storage class*）。第 12 章将介绍存储类别的相关内容，现在只需记住：本章描述的数组属于自动存储类别，意思是这些数组在函数内部声明，且声明时未使用关键字 static。到目前为止，本书所用的变量和数组都是自动存储类别。

在这里提到存储类别的原因是，不同的存储类别有不同的属性，所以不能把本章的内容推广到其他存储类别。对于一些其他存储类别的变量和数组，如果在声明时未初始化，编译器会自动把它们的值设置为 0。

初始化列表中的项数应与数组的大小一致。如果不一致会怎样？我们还是以上一个程序为例，但初始化列表中缺少两个元素，如程序清单 10.3 所示:

程序清单 10.3　somedata.c **程序**

```
/* some_data.c -- 部分初始化数组 */
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int some_data[SIZE] = { 1492, 1066 };
    int i;

    printf("%2s%14s\n",    "i", "some_data[i]");
    for (i = 0; i < SIZE; i++)
        printf("%2d%14d\n", i, some_data[i]);

    return 0;
}
```

下面是该程序的输出：

```
i some_data[i]
0         1492
1         1066
2            0
3            0
```

如上所示，编译器做得很好。当初始化列表中的值少于数组元素个数时，编译器会把剩余的元素都初始化为 0。也就是说，如果不初始化数组，数组元素和未初始化的普通变量一样，其中储存的都是垃圾值；但是，如果部分初始化数组，剩余的元素就会被初始化为 0。

如果初始化列表的项数多于数组元素个数，编译器可没那么仁慈，它会毫不留情地将其视为错误。但是，没必要因此嘲笑编译器。其实，可以省略方括号中的数字，让编译器自动匹配数组大小和初始化列表中的项数（见程序清单 10.4）

程序清单 10.4　day_mon2.c **程序**

```
/* day_mon2.c -- 让编译器计算元素个数 */
#include <stdio.h>
int main(void)
{
    const int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31 };
    int index;

    for (index = 0; index < sizeof days / sizeof days[0]; index++)
        printf("Month %2d has %d days.\n", index + 1, days[index]);

    return 0;
}
```

在程序清单 10.4 中，要注意以下两点。

- 如果初始化数组时省略方括号中的数字，编译器会根据初始化列表中的项数来确定数组的大小。
- 注意 for 循环中的测试条件。由于人工计算容易出错，所以让计算机来计算数组的大小。sizeof 运算符给出它的运算对象的大小（以字节为单位）。所以 sizeof days 是整个数组的大小（以字节为单位），sizeof day[0]是数组中一个元素的大小（以字节为单位）。整个数组的大小除以单个元素的大小就是数组元素的个数。

下面是该程序的输出：

```
Month 1 has 31 days.
Month 2 has 28 days.
Month 3 has 31 days.
Month 4 has 30 days.
Month 5 has 31 days.
Month 6 has 30 days.
Month 7 has 31 days.
Month 8 has 31 days.
Month 9 has 30 days.
Month 10 has 31 days.
```

我们的本意是防止初始化值的个数超过数组的大小，让程序找出数组大小。我们初始化时用了 10 个值，结果就只打印了 10 个值！这就是自动计数的弊端：无法察觉初始化列表中的项数有误。

还有一种初始化数组的方法，但这种方法仅限于初始化字符数组。我们在下一章中介绍。

10.1.2　指定初始化器（C99）

C99 增加了一个新特性：*指定初始化器*（*designated initializer*）。利用该特性可以初始化指定的数组元素。例如，只初始化数组中的最后一个元素。对于传统的 C 初始化语法，必须初始化最后一个元素之前的所有元素，才能初始化它：

```
int arr[6] = {0,0,0,0,0,212}; // 传统的语法
```

而 C99 规定，可以在初始化列表中使用带方括号的下标指明待初始化的元素：

```
int arr[6] = {[5] = 212}; // 把 arr[5]初始化为 212
```

对于一般的初始化，在初始化一个元素后，未初始化的元素都会被设置为 `0`。程序清单 10.5 中的初始化比较复杂。

程序清单 10.5　`designate.c` 程序

```
// designate.c -- 使用指定初始化器
#include <stdio.h>
#define MONTHS 12
int main(void)
{
    int days[MONTHS] = { 31, 28, [4] = 31, 30, 31, [1] = 29 };
    int i;

    for (i = 0; i < MONTHS; i++)
        printf("%2d  %d\n", i + 1, days[i]);

    return 0;
}
```

该程序在支持 C99 的编译器中输出如下：

```
 1   31
 2   29
 3   0
 4   0
 5   31
 6   30
 7   31
 8   0
```

```
9     0
10    0
11    0
12    0
```

以上输出揭示了指定初始化器的两个重要特性。第一，如果指定初始化器后面有更多的值，如该例中的初始化列表中的片段：[4] = 31,30,31，那么后面这些值将被用于初始化指定元素后面的元素。也就是说，在 days[4]被初始化为 31 后，days[5]和 days[6]将分别被初始化为 30 和 31。第二，如果再次初始化指定的元素，那么最后的初始化将会取代之前的初始化。例如，程序清单 10.5 中，初始化列表开始时把 days[1]初始化为 28，但是 days[1]又被后面的指定初始化[1] = 29 初始化为 29。

如果未指定元素大小会怎样？

```
int stuff[] = {1, [6] = 23};        //会发生什么？
int staff[] = {1, [6] = 4, 9, 10};  //会发生什么？
```

编译器会把数组的大小设置为足够装得下初始化的值。所以，stuff 数组有 7 个元素，编号为 0～6；而 staff 数组的元素比 stuff 数组多两个（即有 9 个元素）。

10.1.3　给数组元素赋值

声明数组后，可以借助数组下标（或索引）给数组元素赋值。例如，下面的程序段给数组的所有元素赋值：

```
/* 给数组的元素赋值 */
#include <stdio.h>
#define SIZE 50
int main(void)
{
    int counter, evens[SIZE];

    for (counter = 0; counter < SIZE; counter++)
        evens[counter] = 2 * counter;
    ...
}
```

注意这段代码中使用循环给数组的元素依次赋值。C 不允许把数组作为一个单元赋给另一个数组，除初始化以外也不允许使用花括号列表的形式赋值。下面的代码段演示了一些错误的赋值形式:

```
/* 一些无效的数组赋值 */
#define SIZE 5
int main(void)
{
    int oxen[SIZE] = {5,3,2,8};         /* 初始化没问题 */
    int yaks[SIZE];

    yaks = oxen;                        /* 不允许 */
    yaks[SIZE] = oxen[SIZE];            /* 数组下标越界 */
    yaks[SIZE] = {5,3,2,8};             /* 不起作用 */
```

oxen 数组的最后一个元素是 oxen[SIZE-1]，所以 oxen[SIZE]和 yaks[SIZE]都超出了两个数组的末尾。

10.1.4　数组边界

在使用数组时，要防止数组下标超出边界。也就是说，必须确保下标是有效的值。例如，假设有下面的声明：

```
int doofi[20];
```

那么在使用该数组时，要确保程序中使用的数组下标在 0～19 的范围内，因为编译器不会检查出这种错误（但是，一些编译器发出警告，然后继续编译程序）。

考虑程序清单 10.6 的问题。该程序创建了一个内含 4 个元素的数组，然后错误地使用了-1～6 的下标。

程序清单 10.6　bounds.c 程序

```
// bounds.c -- 数组下标越界
#include <stdio.h>
#define SIZE 4
int main(void)
{
    int value1 = 44;
    int arr[SIZE];
    int value2 = 88;
    int i;

    printf("value1 = %d, value2 = %d\n", value1, value2);
    for (i = -1; i <= SIZE; i++)
        arr[i] = 2 * i + 1;

    for (i = -1; i < 7; i++)
        printf("%2d %d\n", i, arr[i]);
    printf("value1 = %d, value2 = %d\n", value1, value2);
    printf("address of arr[-1]: %p\n", &arr[-1]);
    printf("address of arr[4]: %p\n", &arr[4]);
    printf("address of value1: %p\n", &value1);
    printf("address of value2: %p\n", &value2);

    return 0;
}
```

编译器不会检查数组下标是否使用得当。在 C 标准中，使用越界下标的结果是未定义的。这意味着程序看上去可以运行，但是运行结果很奇怪，或异常中止。下面是使用 GCC 的输出示例：

```
value1 = 44, value2 = 88
-1 -1
 0 1
 1 3
 2 5
 3 7
 4 9
 5 1624678494
 6 32767
value1 = 9, value2 = -1
address of arr[-1]:  0x7fff5fbff8cc
address of arr[4]:   0x7fff5fbff8e0
address of value1:   0x7fff5fbff8e0
address of value2:   0x7fff5fbff8cc
```

注意，该编译器似乎把 value2 储存在数组的前一个位置，把 value1 储存在数组的后一个位置（其他编译器在内存中储存数据的顺序可能不同）。在上面的输出中，arr[-1]与 value2 对应的内存地址相同，arr[4]和 value1 对应的内存地址相同。因此，使用越界的数组下标会导致程序改变其他变量的值。不同的编译器运行该程序的结果可能不同，有些会导致程序异常中止。

C 语言为何会允许这种麻烦事发生？这要归功于 C 信任程序员的原则。不检查边界，C 程序可以运行

更快。编译器没必要捕获所有的下标错误，因为在程序运行之前，数组的下标值可能尚未确定。因此，为安全起见，编译器必须在运行时添加额外代码检查数组的每个下标值，这会降低程序的运行速度。C 相信程序员能编写正确的代码，这样的程序运行速度更快。但并不是所有的程序员都能做到这一点，所以就出现了下标越界的问题。

还要记住一点：数组元素的编号从 0 开始。最好是在声明数组时使用符号常量来表示数组的大小：

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];
    for (i = 0; i < SIZE; i++)
    ....
```

这样做能确保整个程序中的数组大小始终一致。

10.1.5　指定数组的大小

本章前面的程序示例都使用整型常量来声明数组：

```
#define SIZE 4
int main(void)
{
    int arr[SIZE];          // 整数符号常量
    double lots[144];       // 整数字面常量
    ...
```

在 C99 标准之前，声明数组时只能在方括号中使用整型常量表达式。所谓整型常量表达式，是由整型常量构成的表达式。sizeof 表达式被视为整型常量，但是（与 C++不同）const 值不是。另外，表达式的值必须大于 0：

```
int n = 5;
int m = 8;
float a1[5];                // 可以
float a2[5*2 + 1];          //可以
float a3[sizeof(int) + 1];  //可以
float a4[-4];               // 不可以，数组大小必须大于 0
float a5[0];                // 不可以，数组大小必须大于 0
float a6[2.5];              // 不可以，数组大小必须是整数
float a7[(int)2.5];         // 可以，已被强制转换为整型常量
float a8[n];                // C99 之前不允许
float a9[m];                // C99 之前不允许
```

上面的注释表明，以前支持 C90 标准的编译器不允许后两种声明方式。而 C99 标准允许这样声明，这创建了一种新型数组，称为变长数组（*variable-length array*）或简称 VLA（C11 放弃了这一创新的举措，把 VLA 设定为可选，而不是语言必备的特性）。

C99 引入变长数组主要是为了让 C 成为更好的数值计算语言。例如，VLA 简化了把 FORTRAN 现有的数值计算例程库转换为 C 代码的过程。VLA 有一些限制，例如，声明 VLA 时不能进行初始化。在充分了解经典的 C 数组后，我们再详细介绍 VLA。

10.2　多维数组

气象研究员 Tempest Cloud 为完成她的研究项目要分析 5 年内每个月的降水量数据，她首先要解决的问

题是如何表示数据。一个方案是创建 60 个变量，每个变量储存一个数据项（我们曾经提到过这一笨拙的方案，和以前一样，这个方案并不合适）。使用一个内含 60 个元素的数组比将建 60 个变量好，但是如果能把各年的数据分开储存会更好，即创建 5 个数组，每个数组 12 个元素。然而，这样做也很麻烦，如果 Tempest 决定研究 50 年的降水量，岂不是要创建 50 个数组。是否能有更好的方案？

处理这种情况应该使用数组的数组。主数组（*master array*）有 5 个元素（每个元素表示一年），每个元素是内含 12 个元素的数组（每个元素表示一个月）。下面是该数组的声明：

```
float rain[5][12]; // 内含 5 个数组元素的数组，每个数组元素内含 12 个 float 类型的元素
```

理解该声明的一种方法是，先查看中间部分（粗体部分）：

```
float rain[5][12];// rain 是一个内含 5 个元素的数组
```

这说明数组 rain 有 5 个元素，至于每个元素的情况，要查看声明的其余部分（粗体部分）：

```
floatrain[5][12] ;// 一个内含 12 个 float 类型元素的数组
```

这说明每个元素的类型是 float[12]，也就是说，rain 的每个元素本身都是一个内含 12 个 float 类型值的数组。

根据以上分析可知，rain 的首元素 rain[0]是一个内含 12 个 float 类型值的数组。所以，rain[1]、rain[2]等也是如此。如果 rain[0]是一个数组，那么它的首元素就是 rain[0][0]，第 2 个元素是 rain[0][1]，以此类推。简而言之，数组 rain 有 5 个元素，每个元素都是内含 12 个 float 类型元素的数组，rain[0]是内含 12 个 float 值的数组，rain[0][0]是一个 float 类型的值。假设要访问位于 2 行 3 列的值，则使用 rain[2][3]（记住，数组元素的编号从 0 开始，所以 2 行指的是第 3 行）。

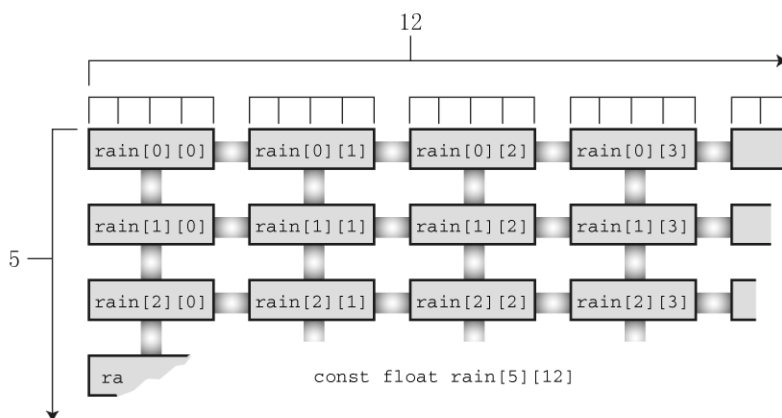


图 10.1 二维数组

该二维视图有助于帮助读者理解二维数组的两个下标。在计算机内部，这样的数组是按顺序储存的，从第 1 个内含 12 个元素的数组开始，然后是第 2 个内含 12 个元素的数组，以此类推。

我们要在气象分析程序中用到这个二维数组。该程序的目标是，计算每年的总降水量、年平均降水量和月平均降水量。要计算年总降水量，必须对一行数据求和；要计算某月份的平均降水量，必须对一列数据求和。二维数组很直观，实现这些操作也很容易。程序清单 10.7 演示了这个程序。

程序清单 10.7 rain.c 程序

```
/* rain.c  -- 计算每年的总降水量、年平均降水量和 5 年中每月的平均降水量 */
#include <stdio.h>
#define MONTHS 12      // 一年的月份数
#define YEARS   5      // 年数
```

```
int main(void)
{
    // 用 2010～2014 年的降水量数据初始化数组
    const float rain[YEARS][MONTHS] =
    {
        { 4.3, 4.3, 4.3, 3.0, 2.0, 1.2, 0.2, 0.2, 0.4, 2.4, 3.5, 6.6 },
        { 8.5, 8.2, 1.2, 1.6, 2.4, 0.0, 5.2, 0.9, 0.3, 0.9, 1.4, 7.3 },
        { 9.1, 8.5, 6.7, 4.3, 2.1, 0.8, 0.2, 0.2, 1.1, 2.3, 6.1, 8.4 },
        { 7.2, 9.9, 8.4, 3.3, 1.2, 0.8, 0.4, 0.0, 0.6, 1.7, 4.3, 6.2 },
        { 7.6, 5.6, 3.8, 2.8, 3.8, 0.2, 0.0, 0.0, 0.0, 1.3, 2.6, 5.2 }
    };
    int year, month;
    float subtot, total;

    printf(" YEAR    RAINFALL  (inches)\n");
    for (year = 0, total = 0; year < YEARS; year++)
    {                        // 每一年，各月的降水量总和
        for (month = 0, subtot = 0; month < MONTHS; month++)
            subtot += rain[year][month];
        printf("%5d %15.1f\n", 2010 + year, subtot);
        total += subtot;   // 5 年的总降水量
    }
    printf("\nThe yearly average is %.1f inches.\n\n", total / YEARS);
    printf("MONTHLY AVERAGES:\n\n");
    printf(" Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct ");
    printf(" Nov  Dec\n");

    for (month = 0; month < MONTHS; month++)
    {                        // 每个月，5 年的总降水量
        for (year = 0, subtot = 0; year < YEARS; year++)
            subtot += rain[year][month];
        printf("%4.1f ", subtot / YEARS);
    }
    printf("\n");

    return 0;
}
```

下面是该程序的输出：

```
YEAR    RAINFALL  (inches)
2010         32.4
2011         37.9
2012         49.8
2013         44.0
2014         32.9

The yearly average is 39.4 inches.

MONTHLY AVERAGES:

Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
7.3 7.3 4.9 3.0 2.3 0.6 1.2 0.3 0.5 1.7 3.6 6.7
```

学习该程序的重点是数组初始化和计算方案。初始化二维数组比较复杂，我们先来看较为简单的计算部分。

程序使用了两个嵌套 for 循环。第 1 个嵌套 for 循环的内层循环，在 year 不变的情况下，遍历 month 计算某年的总降水量；而外层循环，改变 year 的值，重复遍历 month，计算 5 年的总降水量。这种嵌套循环结构常用于处理二维数组，一个循环处理数组的第 1 个下标，另一个循环处理数组的第 2 个下标：

```
for (year = 0, total = 0; year < YEARS; year++)
{ // 处理每一年的数据
    for (month = 0, subtot = 0; month < MONTHS; month++)
    ... // 处理每月的数据
    ... //处理每一年的数据
}
```

第 2 个嵌套 for 循环和第 1 个的结构相同，但是内层循环遍历 year，外层循环遍历 month。记住，每执行一次外层循环，就完整遍历一次内层循环。因此，在改变月份之前，先遍历完年，得到某月 5 年间的平均降水量，以此类推：

```
for (month = 0; month < MONTHS; month++)
{ // 处理每月的数据
    for (year = 0, subtot =0; year < YEARS; year++)
    ... // 处理每年的数据
    ... // 处理每月的数据
}
```

10.2.1　初始化二维数组

初始化二维数组是建立在初始化一维数组的基础上。首先，初始化一维数组如下：

```
sometype ar1[5] = {val1, val2, val3, val4, val5};
```

这里，val1、val2 等表示 sometype 类型的值。例如，如果 sometype 是 int，那么 val1 可能是 7；如果 sometype 是 double，那么 val1 可能是 11.34，诸如此类。但是 rain 是一个内含 5 个元素的数组，每个元素又是内含 12 个 float 类型元素的数组。所以，对 rain 而言，val1 应该包含 12 个值，用于初始化内含 12 个 float 类型元素的一维数组，如下所示：

```
{4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6}
```

也就是说，如果 sometype 是一个内含 12 个 double 类型元素的数组，那么 val1 就是一个由 12 个 double 类型值构成的数值列表。因此，为了初始化二维数组 rain，要用逗号分隔 5 个这样的数值列表：

```
const float rain[YEARS][MONTHS] =
{
    {4.3,4.3,4.3,3.0,2.0,1.2,0.2,0.2,0.4,2.4,3.5,6.6},
    {8.5,8.2,1.2,1.6,2.4,0.0,5.2,0.9,0.3,0.9,1.4,7.3},
    {9.1,8.5,6.7,4.3,2.1,0.8,0.2,0.2,1.1,2.3,6.1,8.4},
    {7.2,9.9,8.4,3.3,1.2,0.8,0.4,0.0,0.6,1.7,4.3,6.2},
    {7.6,5.6,3.8,2.8,3.8,0.2,0.0,0.0,0.0,1.3,2.6,5.2}
};
```

这个初始化使用了 5 个数值列表，每个数值列表都用花括号括起来。第 1 个列表的数据用于初始化数组的第 1 行，第 2 个列表的数据用于初始化数组的第 2 行，以此类推。前面讨论的数据个数和数组大小不匹配的问题同样适用于这里的每一行。也就是说，如果第 1 个列表中只有 10 个数，则只会初始化数组第 1 行的前 10 个元素，而最后两个元素将被默认初始化为 0。如果某列表中的数值个数超出了数组每行的元素个数，则会出错，但是这并不会影响其他行的初始化。

初始化时也可省略内部的花括号，只保留最外面的一对花括号。只要保证初始化的数值个数正确，初始化的效果与上面相同。但是如果初始化的数值不够，则按照先后顺序逐行初始化，直到用完所有的值。后面没有值初始化的元素被统一初始化为 0。图 10.2 演示了这种初始化数组的方法。

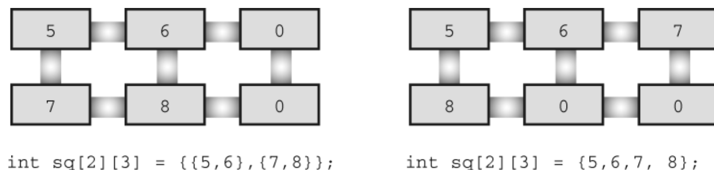


图 10.2　初始化二维数组的两种方法

因为储存在数组 rain 中的数据不能修改，所以程序使用了 const 关键字声明该数组。

10.2.2　其他多维数组

前面讨论的二维数组的相关内容都适用于三维数组或更多维的数组。可以这样声明一个三维数组：

```
int box[10][20][30];
```

可以把一维数组想象成一行数据，把二维数组想象成数据表，把三维数组想象成一叠数据表。例如，把上面声明的三维数组 box 想象成由 10 个二维数组（每个二维数组都是 20 行 30 列）堆叠起来。

还有一种理解 box 的方法是，把 box 看作数组的数组。也就是说，box 内含 10 个元素，每个元素是内含 20 个元素的数组，这 20 个数组元素中的每个元素是内含 30 个元素的数组。或者，可以简单地根据所需的下标值去理解数组。

通常，处理三维数组要使用 3 重嵌套循环，处理四维数组要使用 4 重嵌套循环。对于其他多维数组，以此类推。在后面的程序示例中，我们只使用二维数组。

10.3　指针和数组

第 9 章介绍过指针，指针提供一种以符号形式使用地址的方法。因为计算机的硬件指令非常依赖地址，指针在某种程度上把程序员想要传达的指令以更接近机器的方式表达。因此，使用指针的程序更有效率。尤其是，指针能有效地处理数组。我们很快就会学到，数组表示法其实是在变相地使用指针。

我们举一个变相使用指针的例子：数组名是数组首元素的地址。也就是说，如果 flizny 是一个数组，下面的语句成立：

```
flizny == &flizny[0]; // 数组名是该数组首元素的地址
```

flizny 和&flizny[0]都表示数组首元素的内存地址（&是地址运算符）。两者都是常量，在程序的运行过程中，不会改变。但是，可以把它们赋值给指针变量，然后可以修改指针变量的值，如程序清单 10.8 所示。注意指针加上一个数时，它的值发生了什么变化（转换说明%p 通常以十六进制显示指针的值）。

程序清单 10.8　pnt_add.c 程序

```
// pnt_add.c -- 指针地址
#include <stdio.h>
#define SIZE 4
int main(void)
{
    short dates[SIZE];
    short * pti;
    short index;
    double bills[SIZE];
    double * ptf;
    pti = dates;  // 把数组地址赋给指针
    ptf = bills;
    printf("%23s %15s\n", "short", "double");
```

```
    for (index = 0; index < SIZE; index++)
        printf("pointers + %d: %10p %10p\n", index, pti + index, ptf + index);
    return 0;
}
```

下面是该例的输出示例：

```
                  short          double
pointers + 0: 0x7fff5fbff8dc 0x7fff5fbff8a0
pointers + 1: 0x7fff5fbff8de 0x7fff5fbff8a8
pointers + 2: 0x7fff5fbff8e0 0x7fff5fbff8b0
pointers + 3: 0x7fff5fbff8e2 0x7fff5fbff8b8
```

第 2 行打印的是两个数组开始的地址，下一行打印的是指针加 1 后的地址，以此类推。注意，地址是十六进制的，因此 dd 比 dc 大 1，a1 比 a0 大 1。但是，显示的地址是怎么回事？

0x7fff5fbff8dc + 1 是否是 0x7fff5fbff8de?

0x7fff5fbff8a0 + 1 是否是 0x7fff5fbff8a8?

我们的系统中，地址按字节编址，short 类型占用 2 字节，double 类型占用 8 字节。在 C 中，指针加 1 指的是增加一个*存储单元*。对数组而言，这意味着把加 1 后的地址是下一个*元素*的地址，而不是下一个字节的地址（见图 10.3）。这是为什么必须声明指针所指向对象类型的原因之一。只知道地址不够，因为计算机要知道储存对象需要多少字节（即使指针指向的是标量变量，也要知道变量的类型，否则*pt 就无法正确地取回地址上的值）。

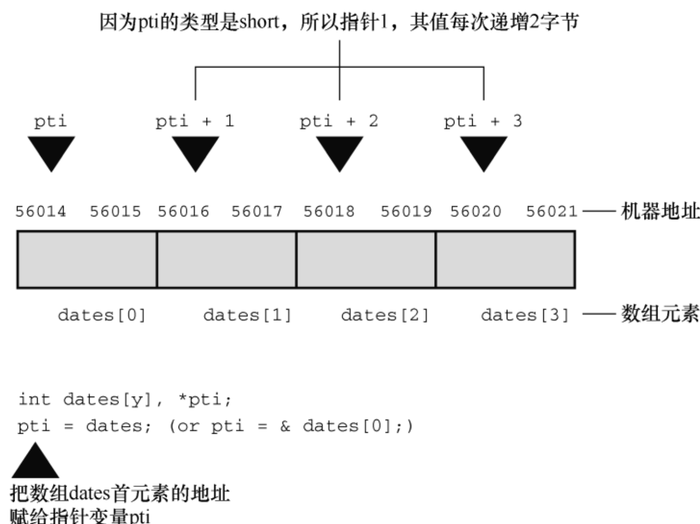


图 10.3　数组和指针加法

现在可以更清楚地定义指向 int 的指针、指向 float 的指针，以及指向其他数据对象的指针。

- 指针的值是它所指向对象的地址。地址的表示方式依赖于计算机内部的硬件。许多计算机（包括 PC 和 Macintosh）都是*按字节编址*，意思是内存中的每个字节都按顺序编号。这里，一个较大对象的地址（如 double 类型的变量）通常是该对象第一个字节的地址。
- 在指针前面使用*运算符可以得到该指针所指向对象的值。
- 指针加 1，指针的值递增它所指向类型的大小（以字节为单位）。

下面的等式体现了 C 语言的灵活性：

```
dates + 2 == &date[2]      // 相同的地址
*(dates + 2) == dates[2]   // 相同的值
```

以上关系表明了数组和指针的关系十分密切，可以使用指针标识数组的元素和获得元素的值。从本质上看，同一个对象有两种表示法。实际上，C 语言标准在描述数组表示法时确实借助了指针。也就是说，定义 ar[n]的意思是*(ar + n)。可以认为*(ar + n)的意思是“到内存的 ar 位置，然后移动 n 个单元，检索储存在那里的值”。

顺带一提，不要混淆 *(dates+2)和*dates+2。间接运算符（*）的优先级高于+，所以*dates+2 相当于(*dates)+2：

```
*(dates + 2)  // dates 第 3 个元素的值
*dates + 2    // dates 第 1 个元素的值加 2
```

明白了数组和指针的关系，便可在编写程序时适时使用数组表示法或指针表示法。运行程序清单 10.9 后输出的结果和程序清单 10.1 输出的结果相同。

程序清单 10.9　day_mon3.c 程序

```
/* day_mon3.c -- uses pointer notation */
#include <stdio.h>
#define MONTHS 12

int main(void)
{
    int days[MONTHS] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
    int index;

    for (index = 0; index < MONTHS; index++)
        printf("Month %2d has %d days.\n", index + 1,
                    *(days + index));   //与 days[index]相同

    return 0;
}
```

这里，days 是数组首元素的地址，days + index 是元素 days[index]的地址，而*(days + index)则是该元素的值，相当于 days[index]。for 循环依次引用数组中的每个元素，并打印各元素的内容。

这样编写程序是否有优势？不一定。编译器编译这两种写法生成的代码相同。程序清单 10.9 要注意的是，指针表示法和数组表示法是两种等效的方法。该例演示了可以用指针表示数组，反过来，也可以用数组表示指针。在使用以数组为参数的函数时要注意这点。

10.4　函数、数组和指针

假设要编写一个处理数组的函数，该函数返回数组中所有元素之和，待处理的是名为 marbles 的 int 类型数组。应该如何调用该函数？也许是下面这样：

```
total = sum(marbles); // 可能的函数调用
```

那么，该函数的原型是什么？记住，数组名是该数组首元素的地址，所以实际参数 marbles 是一个储存 int 类型值的地址，应把它赋给一个指针形式参数，即该形参是一个指向 int 的指针：

```
int sum(int * ar);// 对应的函数原型
```

sum()从该参数获得了什么信息？它获得了该数组首元素的地址，知道要在该位置上找出一个整数。

注意，该参数并未包含数组元素个数的信息。我们有两种方法让函数获得这一信息。第一种方法是，在函数代码中写上固定的数组大小：

```
int sum(int * ar) // 相应的函数定义
{
     int i;
     int total = 0;

     for (i = 0; i < 10; i++)    // 假设数组有10个元素
          total += ar[i];        // ar[i] 与 *(ar + i) 相同
     return total;
}
```

既然能使用指针表示数组名，也可以用数组名表示指针。另外，回忆一下，+=运算符把右侧运算对象加到左侧运算对象上。因此，total 是当前数组元素之和。

该函数定义有限制，只能计算 10 个 int 类型的元素。另一个比较灵活的方法是把数组大小作为第 2 个参数：

```
int sum(int * ar, int n)        // 更通用的方法
{
     int i;
     int total = 0;

     for (i = 0; i < n; i++)     // 使用 n 个元素
          total += ar[i];        // ar[i] 和 *(ar + i) 相同
     return total;
}
```

这里，第 1 个形参告诉函数该数组的地址和数据类型，第 2 个形参告诉函数该数组中元素的个数。

关于函数的形参，还有一点要注意。只有在函数原型或函数定义头中，才可以用 int ar[]代替 int * ar：

```
int sum (int ar[], int n);
```

int *ar 形式和 int ar[]形式都表示 ar 是一个指向 int 的指针。但是，int ar[]只能用于声明形式参数。第 2 种形式（int ar[]）提醒读者指针 ar 指向的不仅仅一个 int 类型值，还是一个 int 类型数组的元素。

注意　声明数组形参

因为数组名是该数组首元素的地址，作为实际参数的数组名要求形式参数是一个与之匹配的指针。只有在这种情况下，C 才会把 int ar[]和 int * ar 解释成一样。也就是说，ar 是指向 int 的指针。由于函数原型可以省略参数名，所以下面 4 种原型都是等价的：

```
int sum(int *ar, int n);
int sum(int *, int);
int sum(int ar[], int n);
int sum(int [], int);
```

但是，在函数定义中不能省略参数名。下面两种形式的函数定义等价：

```
int sum(int *ar, int n)
{
    // 其他代码已省略
}

int sum(int ar[], int n);
{
```

```
        //其他代码已省略
    }
```

可以使用以上提到的任意一种函数原型和函数定义。

程序清单 10.10 演示了一个程序，使用 `sum()`函数。该程序打印原始数组的大小和表示该数组的函数形参的大小（如果你的编译器不支持用转换说明`%zd`打印 `sizeof` 返回值，可以用`%u` 或`%lu` 来代替）。

程序清单 10.10　`sum_arr1.c` 程序

```
// sum_arr1.c -- 数组元素之和
// 如果编译器不支持 %zd，用 %u 或 %lu 替换它
#include <stdio.h>
#define SIZE 10
int sum(int ar[], int n);
int main(void)
{
    int marbles[SIZE] = { 20, 10, 5, 39, 4, 16, 19, 26, 31, 20 };
    long answer;

    answer = sum(marbles, SIZE);
    printf("The total number of marbles is %ld.\n", answer);
    printf("The size of marbles is %zd bytes.\n",
        sizeof marbles);

    return 0;
}

int sum(int ar[], int n)    // 这个数组的大小是?
{
    int i;
    int total = 0;

    for (i = 0; i < n; i++)
        total += ar[i];
    printf("The size of ar is %zd bytes.\n", sizeof ar);

    return total;
}
```

该程序的输出如下：

```
The size of ar is 8 bytes.
The total number of marbles is 190.
The size of marbles is 40 bytes.
```

注意，`marbles` 的大小是 40 字节。这没问题，因为 `marbles` 内含 10 个 `int` 类型的值，每个值占 4 字节，所以整个 `marbles` 的大小是 40 字节。但是，`ar` 才 8 字节。这是因为 `ar` 并不是数组本身，它是一个指向 `marbles` 数组首元素的指针。我们的系统中用 8 字节储存地址，所以指针变量的大小是 8 字节（其他系统中地址的大小可能不是 8 字节）。简而言之，在程序清单 10.10 中，`marbles` 是一个数组，`ar` 是一个指向 `marbles` 数组首元素的指针，利用 C 中数组和指针的特殊关系，可以用数组表示法来表示指针 `ar`。

10.4.1　使用指针形参

函数要处理数组必须知道何时开始、何时结束。sum()函数使用一个指针形参标识数组的开始，用一个整数形参表明待处理数组的元素个数（指针形参也表明了数组中的数据类型）。但是这并不是给函数传递必备信息的唯一方法。还有一种方法是传递两个指针，第 1 个指针指明数组的开始处（与前面用法相同），第 2 个指针指明数组的结束处。程序清单 10.11 演示了这种方法，同时该程序也表明了指针形参是变量，这意味着可以用索引表明访问数组中的哪一个元素。

程序清单 10.11　sum_arr2.c **程序**

```
/* sum_arr2.c -- 数组元素之和 */
#include <stdio.h>
#define SIZE 10
int sump(int * start, int * end);
int main(void)
{
    int marbles[SIZE] = { 20, 10, 5, 39, 4, 16, 19, 26, 31, 20 };
    long answer;

    answer = sump(marbles, marbles + SIZE);
    printf("The total number of marbles is %ld.\n", answer);

    return 0;
}

/* 使用指针算法 */
int sump(int * start, int * end)
{
    int total = 0;

    while (start < end)
    {
        total += *start;    // 把数组元素的值加起来
        start++;            // 让指针指向下一个元素
    }

    return total;
}
```

指针 start 开始指向 marbles 数组的首元素，所以赋值表达式 total += *start 把首元素（20）加给 total。然后，表达式 start++递增指针变量 start，使其指向数组的下一个元素。因为 start 是指向 int 的指针，start 递增 1 相当于其值递增 int 类型的大小。

注意，sump()函数用另一种方法结束加法循环。sum()函数把元素的个数作为第 2 个参数，并把该参数作为循环测试的一部分：

```
for( i = 0; i < n; i++)
```

而 sump()函数则使用第 2 个指针来结束循环：

```
while (start < end)
```

因为 while 循环的测试条件是一个不相等的关系，所以循环最后处理的一个元素是 end 所指向位置的前一个元素。这意味着 end 指向的位置实际上在数组最后一个元素的后面。C 保证在给数组分配空间时，

指向数组后面第一个位置的指针仍是有效的指针。这使得 while 循环的测试条件是有效的，因为 start 在循环中最后的值是 end[1]。注意，使用这种“越界”指针的函数调用更为简洁：

```
answer = sump(marbles, marbles + SIZE);
```

因为下标从 0 开始，所以 marbles + SIZE 指向数组末尾的下一个位置。如果 end 指向数组的最后一个元素而不是数组末尾的下一个位置，则必须使用下面的代码：

```
answer = sump(marbles, marbles + SIZE - 1);
```

这种写法既不简洁也不好记，很容易导致编程错误。顺带一提，虽然 C 保证了 marbles + SIZE 有效，但是对 marbles[SIZE]（即储存在该位置上的值）未作任何保证，所以程序不能访问该位置。

还可以把循环体压缩成一行代码：

```
total += *start++;
```

一元运算符*和++的优先级相同，但结合律是从右往左，所以 start++先求值，然后才是*start。也就是说，指针 start 先递增后指向。使用后缀形式（即 start++而不是++start）意味着先把指针指向位置上的值加到 total 上，然后再递增指针。如果使用*++start，顺序则反过来，先递增指针，再使用指针指向位置上的值。如果使用(*start)++，则先使用 start 指向的值，再递增该值，而不是递增指针。这样，指针将一直指向同一个位置，但是该位置上的值发生了变化。虽然*start++的写法比较常用，但是*(start++)这样写更清楚。程序清单 10.12 的程序演示了这些优先级的情况。

程序清单 10.12　order.c 程序

```
/* order.c -- 指针运算中的优先级 */
#include <stdio.h>
int data[2] = { 100, 200 };
int moredata[2] = { 300, 400 };
int main(void)
{
    int * p1, *p2, *p3;

    p1 = p2 = data;
    p3 = moredata;
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",*p1, *p2, *p3);
    printf("*p1++ = %d, *++p2 = %d, (*p3)++ = %d\n",*p1++, *++p2, (*p3)++);
    printf("  *p1 = %d,   *p2 = %d,     *p3 = %d\n",*p1, *p2, *p3);

    return 0;
}
```

下面是该程序的输出：

```
  *p1 = 100,   *p2 = 100,     *p3 = 300
*p1++ = 100, *++p2 = 200,  (*p3)++ = 300
  *p1 = 200,   *p2 = 200,     *p3 = 301
```

只有(*p3)++改变了数组元素的值，其他两个操作分别把 p1 和 p2 指向数组的下一个元素。

10.4.2　指针表示法和数组表示法

从以上分析可知，处理数组的函数实际上用指针作为参数，但是在编写这样的函数时，可以选择是使用数组表示法还是指针表示法。如程序清单 10.10 所示，使用数组表示法，让函数是处理数组的这一意图更

[1] 在最后一次 while 循环中执行完 start++;后，start 的值就是 end 的值。——译者注

加明显。另外，许多其他语言的程序员对数组表示法更熟悉，如 FORTRAN、Pascal、Modula-2 或 BASIC。其他程序员可能更习惯使用指针表示法，觉得使用指针更自然，如程序清单 10.11 所示。

至于 C 语言，ar[i]和*(ar+1)这两个表达式都是等价的。无论 ar 是数组名还是指针变量，这两个表达式都没问题。但是，只有当 ar 是指针变量时，才能使用 ar++这样的表达式。

指针表示法（尤其与递增运算符一起使用时）更接近机器语言，因此一些编译器在编译时能生成效率更高的代码。然而，许多程序员认为他们的主要任务是确保代码正确、逻辑清晰，而代码优化应该留给编译器去做。

10.5　指针操作

可以对指针进行哪些操作？C 提供了一些基本的指针操作，下面的程序示例中演示了 8 种不同的操作。为了显示每种操作的结果，该程序打印了指针的值（该指针指向的地址）、储存在指针指向地址上的值，以及指针自己的地址。如果编译器不支持%p 转换说明，可以用%u 或%lu 代替%p；如果编译器不支持用%td 转换说明打印地址的差值，可以用%d 或%ld 来代替。

程序清单 10.13 演示了指针变量的 8 种基本操作。除了这些操作，还可以使用关系运算符来比较指针。

程序清单 10.13　ptr_ops.c 程序

```
// ptr_ops.c -- 指针操作
#include <stdio.h>
int main(void)
{
    int urn[5] = { 100, 200, 300, 400, 500 };
    int * ptr1, *ptr2, *ptr3;

    ptr1 = urn;              // 把一个地址赋给指针
    ptr2 = &urn[2];          // 把一个地址赋给指针
                             // 解引用指针，以及获得指针的地址
    printf("pointer value, dereferenced pointer, pointer address:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n", ptr1, *ptr1, &ptr1);

    // 指针加法
    ptr3 = ptr1 + 4;
    printf("\nadding an int to a pointer:\n");
    printf("ptr1 + 4 = %p, *(ptr1 + 4) = %d\n", ptr1 + 4, *(ptr1 + 4));
    ptr1++;                  // 递增指针
    printf("\nvalues after ptr1++:\n");
    printf("ptr1 = %p, *ptr1 =%d, &ptr1 = %p\n", ptr1, *ptr1, &ptr1);
    ptr2--;                  // 递减指针
    printf("\nvalues after --ptr2:\n");
    printf("ptr2 = %p, *ptr2 = %d, &ptr2 = %p\n", ptr2, *ptr2, &ptr2);
    --ptr1;                  // 恢复为初始值
    ++ptr2;                  // 恢复为初始值
    printf("\nPointers reset to original values:\n");
    printf("ptr1 = %p, ptr2 = %p\n", ptr1, ptr2);
    // 一个指针减去另一个指针
    printf("\nsubtracting one pointer from another:\n");
    printf("ptr2 = %p, ptr1 = %p, ptr2 - ptr1 = %td\n", ptr2, ptr1, ptr2 - ptr1);
```

```
    // 一个指针减去一个整数
    printf("\nsubtracting an int from a pointer:\n");
    printf("ptr3 = %p, ptr3 - 2 = %p\n", ptr3, ptr3 - 2);

    return 0;
}
```

下面是我们的系统运行该程序后的输出：

```
pointer value, dereferenced pointer, pointer address:
ptr1 = 0x7fff5fbff8d0, *ptr1 =100, &ptr1 = 0x7fff5fbff8c8

adding an int to a pointer:
ptr1 + 4 = 0x7fff5fbff8e0, *(ptr1 + 4) = 500

values after ptr1++:
ptr1 = 0x7fff5fbff8d4, *ptr1 =200, &ptr1 = 0x7fff5fbff8c8

values after --ptr2:
ptr2 = 0x7fff5fbff8d4, *ptr2 = 200, &ptr2 = 0x7fff5fbff8c0

Pointers reset to original values:
ptr1 = 0x7fff5fbff8d0, ptr2 = 0x7fff5fbff8d8

subtracting one pointer from another:
ptr2 = 0x7fff5fbff8d8, ptr1 = 0x7fff5fbff8d0, ptr2 - ptr1 = 2

subtracting an int from a pointer:
ptr3 = 0x7fff5fbff8e0, ptr3 - 2 = 0x7fff5fbff8d8
```

下面分别描述了指针变量的基本操作。

- **赋值**：可以把地址赋给指针。例如，用数组名、带地址运算符（&）的变量名、另一个指针进行赋值。在该例中，把 urn 数组的首地址赋给了 ptr1，该地址的编号恰好是 0x7fff5fbff8d0。变量 ptr2 获得数组 urn 的第 3 个元素（urn[2]）的地址。注意，地址应该和指针类型兼容。也就是说，不能把 double 类型的地址赋给指向 int 的指针，至少要避免不明智的类型转换。C99/C11 已经强制不允许这样做。
- **解引用**：*运算符给出指针指向地址上储存的值。因此，*ptr1 的初值是 100，该值储存在编号为 0x7fff5fbff8d0 的地址上。
- **取址**：和所有变量一样，指针变量也有自己的地址和值。对指针而言，&运算符给出指针本身的地址。本例中，ptr1 储存在内存编号为 0x7fff5fbff8c8 的地址上，该存储单元储存的内容是 0x7fff5fbff8d0，即 urn 的地址。因此&ptr1 是指向 ptr1 的指针，而 ptr1 是指向 utn[0] 的指针。
- **指针与整数相加**：可以使用+运算符把指针与整数相加，或整数与指针相加。无论哪种情况，整数都会和指针所指向类型的大小（以字节为单位）相乘，然后把结果与初始地址相加。因此 ptr1 + 4 与&urn[4]等价。如果相加的结果超出了初始指针指向的数组范围，计算结果则是未定义的。除非正好超过数组末尾第一个位置，C 保证该指针有效。
- **递增指针**：递增指向数组元素的指针可以让该指针移动至数组的下一个元素。因此，ptr1++相当于把 ptr1 的值加上 4（我们的系统中 int 为 4 字节），ptr1 指向 urn[1]（见图 10.4，该图中使用了简化的地址）。现在 ptr1 的值是 0x7fff5fbff8d4（数组的下一个元素的地址），*ptr 的值为

200（即 urn[1]的值）。注意，ptr1 本身的地址仍是 0x7fff5fbff8c8。毕竟，变量不会因为值发生变化就移动位置。

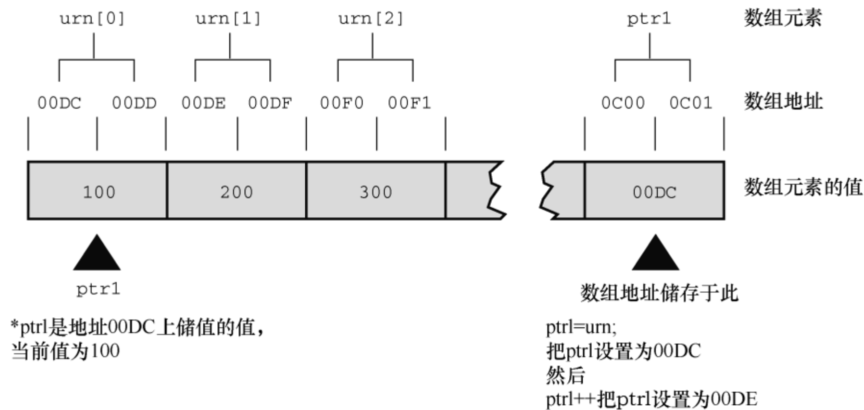


图 10.4　递增指向 int 的指针

- **指针减去一个整数**：可以使用-运算符从一个指针中减去一个整数。指针必须是第 1 个运算对象，整数是第 2 个运算对象。该整数将乘以指针指向类型的大小（以字节为单位），然后用初始地址减去乘积。所以 ptr3 - 2 与&urn[2]等价，因为 ptr3 指向的是&arn[4]。如果相减的结果超出了初始指针所指向数组的范围，计算结果则是未定义的。除非正好超过数组末尾第一个位置，C 保证该指针有效。
- **递减指针**：当然，除了递增指针还可以递减指针。在本例中，递减 ptr3 使其指向数组的第 2 个元素而不是第 3 个元素。前缀或后缀的递增和递减运算符都可以使用。注意，在重置 ptr1 和 ptr2 前，它们都指向相同的元素 urn[1]。
- **指针求差**：可以计算两个指针的差值。通常，求差的两个指针分别指向同一个数组的不同元素，通过计算求出两元素之间的距离。差值的单位与数组类型的单位相同。例如，程序清单 10.13 的输出中，ptr2 - ptr1 得 2，意思是这两个指针所指向的两个元素相隔两个 int，而不是 2 字节。只要两个指针都指向相同的数组（或者其中一个指针指向数组后面的第 1 个地址），C 都能保证相减运算有效。如果指向两个不同数组的指针进行求差运算可能会得出一个值，或者导致运行时错误。
- **比较**：使用关系运算符可以比较两个指针的值，前提是两个指针都指向相同类型的对象。

注意，这里的减法有两种。可以用一个指针减去另一个指针得到一个整数，或者用一个指针减去一个整数得到另一个指针。

在递增或递减指针时还要注意一些问题。编译器不会检查指针是否仍指向数组元素。C 只能保证指向数组任意元素的指针和指向数组后面第 1 个位置的指针有效。但是，如果递增或递减一个指针后超出了这个范围，则是未定义的。另外，可以解引用指向数组任意元素的指针。但是，即使指针指向数组后面一个位置是有效的，也能解引用这样的越界指针。

解引用未初始化的指针

说到注意事项，一定要牢记一点：千万不要解引用未初始化的指针。例如，考虑下面的例子：

```
int * pt; // 未初始化的指针
*pt = 5;        // 严重的错误
```

为何不行？第 2 行的意思是把 5 储存在 pt 指向的位置。但是 pt 未被初始化，其值是一个随机值，

> 所以不知道 5 将储存在何处。这可能不会出什么错，也可能会擦写数据或代码，或者导致程序崩溃。切记：创建一个指针时，系统只分配了储存指针本身的内存，并未分配储存数据的内存。因此，在使用指针之前，必须先用已分配的地址初始化它。例如，可以用一个现有变量的地址初始化该指针（使用带指针形参的函数时，就属于这种情况）。或者还可以使用第 12 章将介绍的 malloc()函数先分配内存。无论如何，使用指针时一定要注意，不要解引用未初始化的指针！
>
> ```
> double * pd;  // 未初始化的指针
> *pd = 2.4;    // 不要这样做
> ```

假设

```
int urn[3];
int * ptr1, * ptr2;
```

下面是一些有效和无效的语句：

有效语句	无效语句
ptr1++;	urn++;
ptr2 = ptr1 + 2;	ptr2 = ptr2 + ptr1;
ptr2 = urn + 1;	ptr2 = urn * ptr1;

基于这些有效的操作，C 程序员创建了指针数组、函数指针、指向指针的指针数组、指向函数的指针数组等。别紧张，接下来我们将根据已学的内容介绍指针的一些基本用法。指针的第 1 个基本用法是在函数间传递信息。前面学过，如果希望在被调函数中改变主调函数的变量，必须使用指针。指针的第 2 个基本用法是用在处理数组的函数中。下面我们再来看一个使用函数和数组的编程示例。

10.6　保护数组中的数据

编写一个处理基本类型（如，int）的函数时，要选择是传递 int 类型的值还是传递指向 int 的指针。通常都是直接传递数值，只有程序需要在函数中改变该数值时，才会传递指针。对于数组别无选择，必须传递指针，因为这样做效率高。如果一个函数按值传递数组，则必须分配足够的空间来储存原数组的副本，然后把原数组所有的数据拷贝至新的数组中。如果把数组的地址传递给函数，让函数直接处理原数组则效率要高。

传递地址会导致一些问题。C 通常都按值传递数据，因为这样做可以保证数据的完整性。如果函数使用的是原始数据的副本，就不会意外修改原始数据。但是，处理数组的函数通常都需要使用原始数据，因此这样的函数可以修改原数组。有时，这正是我们需要的。例如，下面的函数给数组的每个元素都加上一个相同的值：

```
void add_to(double ar[], int n, double val)
{
     int i;
     for (i = 0; i < n; i++)
          ar[i] += val;
}
```

因此，调用该函数后，prices 数组中的每个元素的值都增加了 2.5：

```
add_to(prices, 100, 2.50);
```

该函数修改了数组中的数据。之所以可以这样做，是因为函数通过指针直接使用了原始数据。

然而，其他函数并不需要修改数据。例如，下面的函数计算数组中所有元素之和，它不用改变数组的数据。但是，由于 ar 实际上是一个指针，所以编程错误可能会破坏原始数据。例如，下面示例中的 ar[i]++ 会导致数组中每个元素的值都加 1：

```
int sum(int ar[], int n)   // 错误的代码
```

```
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i]++;  // 错误递增了每个元素的值
    return total;
}
```

10.6.1　对形式参数使用 const

在 K&R C 的年代，避免类似错误的唯一方法是提高警惕。ANSI C 提供了一种预防手段。如果函数的意图不是修改数组中的数据内容，那么在函数原型和函数定义中声明形式参数时应使用关键字 const。例如，sum()函数的原型和定义如下：

```
int sum(const int ar[], int n); /* 函数原型 */

int sum(const int ar[], int n) /* 函数定义 */
{
    int i;
    int total = 0;

    for( i = 0; i < n; i++)
        total += ar[i];
    return total;
}
```

以上代码中的 const 告诉编译器，该函数不能修改 ar 指向的数组中的内容。如果在函数中不小心使用类似 ar[i]++的表达式，编译器会捕获这个错误，并生成一条错误信息。

这里一定要理解，这样使用 const 并不是要求原数组是常量，而是该函数在处理数组时将其视为常量，不可更改。这样使用 const 可以保护数组的数据不被修改，就像按值传递可以保护基本数据类型的原始值不被改变一样。一般而言，如果编写的函数需要修改数组，在声明数组形参时则不使用 const；如果编写的函数不用修改数组，那么在声明数组形参时最好使用 const。

程序清单 10.14 的程序中，一个函数显示数组的内容，另一个函数给数组每个元素都乘以一个给定值。因为第 1 个函数不用改变数组，所以在声明数组形参时使用了 const；而第 2 个函数需要修改数组元素的值，所以不使用 const。

程序清单 10.14　arf.c 程序

```
/* arf.c -- 处理数组的函数 */
#include <stdio.h>
#define SIZE 5
void show_array(const double ar[], int n);
void mult_array(double ar[], int n, double mult);
int main(void)
{
    double dip[SIZE] = { 20.0, 17.66, 8.2, 15.3, 22.22 };

    printf("The original dip array:\n");
    show_array(dip, SIZE);
    mult_array(dip, SIZE, 2.5);
    printf("The dip array after calling mult_array():\n");
    show_array(dip, SIZE);
```

```
    return 0;
}

/* 显示数组的内容 */
void show_array(const double ar[], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("%8.3f ", ar[i]);
    putchar('\n');
}

/* 把数组的每个元素都乘以相同的值 */
void mult_array(double ar[], int n, double mult)
{
    int i;

    for (i = 0; i < n; i++)
        ar[i] *= mult;
}
```

下面是该程序的输出：

```
The original dip array:
  20.000  17.660   8.200   15.300  22.220
The dip array after calling mult_array():
  50.000  44.150   20.500  38.250  55.550
```

注意该程序中两个函数的返回类型都是 void。虽然 mult_array()函数更新了 dip 数组的值，但是并未使用 return 机制。

10.6.2　const 的其他内容

我们在前面使用 const 创建过变量：

```
const double PI = 3.14159;
```

虽然用#define 指令可以创建类似功能的符号常量，但是 const 的用法更加灵活。可以创建 const 数组、const 指针和指向 const 的指针。

程序清单 10.4 演示了如何使用 const 关键字保护数组：

```
#define MONTHS 12
...
const int days[MONTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

如果程序稍后尝试改变数组元素的值，编译器将生成一个编译期错误消息：

```
days[9] = 44;      /* 编译错误 */
```

指向 const 的指针不能用于改变值。考虑下面的代码：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * pd = rates;      // pd 指向数组的首元素
```

第 2 行代码把 pd 指向的 double 类型的值声明为 const，这表明不能使用 pd 来更改它所指向的值：

```
*pd = 29.89;        // 不允许
pd[2] = 222.22;     //不允许
rates[0] = 99.99; // 允许，因为 rates 未被 const 限定
```

无论是使用指针表示法还是数组表示法，都不允许使用 pd 修改它所指向数据的值。但是要注意，因为 rates 并未被声明为 const，所以仍然可以通过 rates 修改元素的值。另外，可以让 pd 指向别处：

```
pd++; /* 让pd指向rates[1] -- 没问题 */
```

指向 const 的指针通常用于函数形参中，表明该函数不会使用指针改变数据。例如，程序清单 10.14 中的 show_array()函数原型如下：

```
void show_array(const double *ar, int n);
```

关于指针赋值和 const 需要注意一些规则。首先，把 const 数据或非 const 数据的地址初始化为指向 const 的指针或为其赋值是合法的：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
const double * pc = rates; // 有效
pc = locked;               //有效
pc = &rates[3];            //有效
```

然而，只能把非 const 数据的地址赋给普通指针：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double locked[4] = {0.08, 0.075, 0.0725, 0.07};
double * pnc = rates;  // 有效
pnc = locked;          // 无效
pnc = &rates[3];       // 有效
```

这个规则非常合理。否则，通过指针就能改变 const 数组中的数据。

应用以上规则的例子，如 show_array()函数可以接受普通数组名和 const 数组名作为参数，因为这两种参数都可以用来初始化指向 const 的指针：

```
show_array(rates, 5);      // 有效
show_array(locked, 4);     // 有效
```

因此，对函数的形参使用 const 不仅能保护数据，还能让函数处理 const 数组。

另外，不应该把 const 数组名作为实参传递给 mult_array()这样的函数：

```
mult_array(rates, 5, 1.2);      // 有效
mult_array(locked, 4, 1.2);     // 不要这样做
```

C 标准规定，使用非 const 标识符（如，mult_arry()的形参 ar）修改 const 数据（如，locked）导致的结果是未定义的。

const 还有其他的用法。例如，可以声明并初始化一个不能指向别处的指针，关键是 const 的位置：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
double * const pc = rates; // pc指向数组的开始
pc = &rates[2];            // 不允许，因为该指针不能指向别处
*pc = 92.99;               // 没问题 -- 更改rates[0]的值
```

可以用这种指针修改它所指向的值，但是它只能指向初始化时设置的地址。

最后，在创建指针时还可以使用 const 两次，该指针既不能更改它所指向的地址，也不能修改指向地址上的值：

```
double rates[5] = {88.99, 100.12, 59.45, 183.11, 340.5};
const double * const pc = rates;
pc = &rates[2];     //不允许
*pc = 92.99;        //不允许
```

10.7　指针和多维数组

指针和多维数组有什么关系？为什么要了解它们的关系？处理多维数组的函数要用到指针，所以在使用这种函数之前，先要更深入地学习指针。至于第 1 个问题，我们通过几个示例来回答。为简化讨论，我们使用较小的数组。假设有下面的声明：

```
int zippo[4][2]; /* 内含 int 数组的数组 */
```

然后数组名 zippo 是该数组首元素的地址。在本例中，zippo 的首元素是一个内含两个 int 值的数组，所以 zippo 是这个内含两个 int 值的数组的地址。下面，我们从指针的属性进一步分析。

- 因为 zippo 是数组首元素的地址，所以 zippo 的值和&zippo[0]的值相同。而 zippo[0]本身是一个内含两个整数的数组,所以 zippo[0]的值和它首元素(一个整数)的地址(即&zippo[0][0]的值）相同。简而言之，zippo[0]是一个占用一个 int 大小对象的地址，而 zippo 是一个占用两个 int 大小对象的地址。由于这个整数和内含两个整数的数组都开始于同一个地址,所以 zippo 和 zippo[0]的值相同。
- 给指针或地址加 1，其值会增加对应类型大小的数值。在这方面，zippo 和 zippo[0]不同，因为 zippo 指向的对象占用了两个 int 大小，而 zippo[0]指向的对象只占用一个 int 大小。因此，zippo + 1 和 zippo[0] + 1 的值不同。
- 解引用一个指针（在指针前使用*运算符）或在数组名后使用带下标的[]运算符，得到引用对象代表的值。因为 zippo[0]是该数组首元素（zippo[0][0]）的地址，所以*(zippo[0])表示储存在 zippo[0][0]上的值(即一个 int 类型的值)。与此类似,*zippo 代表该数组首元素(zippo[0])的值,但是 zippo[0]本身是一个 int 类型值的地址。该值的地址是&zippo[0][0],所以*zippo 就是&zippo[0][0]。对两个表达式应用解引用运算符表明，**zippo 与*&zippo[0][0]等价，这相当于 zippo[0][0]，即一个 int 类型的值。简而言之，zippo 是地址的地址，必须解引用两次才能获得原始值。地址的地址或指针的指针是就是*双重间接*（*double indirection*）的例子。

显然，增加数组维数会增加指针的复杂度。现在，大部分初学者都开始意识到指针为什么是 C 语言中最难的部分。认真思考上述内容，看看是否能用所学的知识解释程序清单 10.15 中的程序。该程序显示了一些地址值和数组的内容。

程序清单 10.15　zippo1.c 程序

```
/* zippo1.c --  zippo 的相关信息 */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };

    printf("   zippo = %p,    zippo + 1 = %p\n",zippo, zippo + 1);
    printf("zippo[0] = %p, zippo[0] + 1 = %p\n",zippo[0], zippo[0] + 1);
    printf("  *zippo = %p,   *zippo + 1 = %p\n",*zippo, *zippo + 1);
    printf("zippo[0][0] = %d\n", zippo[0][0]);
    printf("  *zippo[0] = %d\n", *zippo[0]);
    printf("    **zippo = %d\n", **zippo);
    printf("      zippo[2][1] = %d\n", zippo[2][1]);
    printf("*(*(zippo+2) + 1) = %d\n", *(*(zippo + 2) + 1));

    return 0;
}
```

下面是我们的系统运行该程序后的输出：

```
   zippo = 0x0064fd38,      zippo + 1 = 0x0064fd40
 zippo[0]= 0x0064fd38,   zippo[0] + 1 = 0x0064fd3c
  *zippo = 0x0064fd38,     *zippo + 1 = 0x0064fd3c
zippo[0][0] = 2
  *zippo[0] = 2
    **zippo = 2
      zippo[2][1] = 3
*(*(zippo+2) + 1) = 3
```

其他系统显示的地址值和地址形式可能不同，但是地址之间的关系与以上输出相同。该输出显示了二维数组 zippo 的地址和一维数组 zippo[0]的地址相同。它们的地址都是各自数组首元素的地址，因而与&zippo[0][0]的值也相同。

尽管如此，它们也有差别。在我们的系统中，int 是 4 字节。前面讨论过，zippo[0]指向一个 4 字节的数据对象。zippo[0]加 1，其值加 4（十六进制中，38+4 得 3c）。数组名 zippo 是一个内含 2 个 int 类型值的数组的地址，所以 zippo 指向一个 8 字节的数据对象。因此，zippo 加 1，它所指向的地址加 8 字节（十六进制中，38+8 得 40）。

该程序演示了 zippo[0]和*zippo 完全相同，实际上确实如此。然后，对二维数组名解引用两次，得到储存在数组中的值。使用两个间接运算符（*）或者使用两对方括号（[]）都能获得该值（还可以使用一个*和一对[]，但是我们暂不讨论这么多情况）。

要特别注意，与 zippo[2][1]等价的指针表示法是*(*(zippo+2) + 1)。看上去比较复杂，应最好能理解。下面列出了理解该表达式的思路：

```
zippo              ←二维数组首元素的地址（每个元素都是内含两个 int 类型元素的一维数组）
zippo+2            ←二维数组的第 3 个元素（即一维数组）的地址
*(zippo+2)         ←二维数组的第 3 个元素（即一维数组）的首元素（一个 int 类型的值）地址
*(zippo+2) + 1     ←二维数组的第 3 个元素（即一维数组）的第 2 个元素（也是一个 int 类型的值）地址
*(*(zippo+2) + 1)  ←二维数组的第 3 个一维数组元素的第 2 个 int 类型元素的值，即数组的第 3 行第 2
                     列的值（zippo[2][1]）
```

以上分析并不是为了说明用指针表示法（*(*(zippo+2) + 1)）代替数组表示法（zippo[2][1]），而是提示读者，如果程序恰巧使用一个指向二维数组的指针，而且要通过该指针获取值时，最好用简单的数组表示法，而不是指针表示法。

图 10.5 以另一种视图演示了数组地址、数组内容和指针之间的关系。

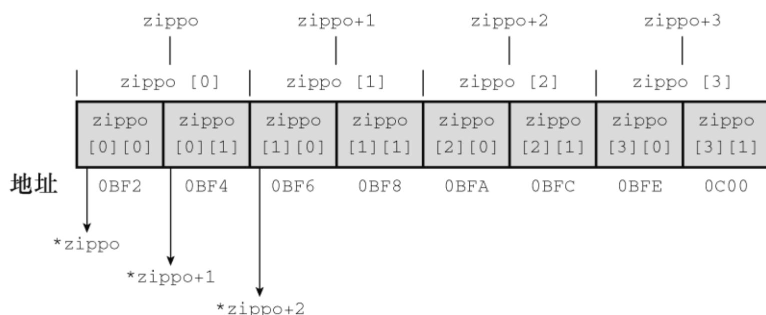


图 10.5　数组的数组

10.7.1　指向多维数组的指针

如何声明一个指针变量 pz 指向一个二维数组（如，zippo）？在编写处理类似 zippo 这样的二维数组时会用到这样的指针。把指针声明为指向 int 的类型还不够。因为指向 int 只能与 zippo[0]的类型匹配，说明该指针指向一个 int 类型的值。但是 zippo 是它首元素的地址，该元素是一个内含两个 int 类型值的一维数组。因此，pz 必须指向一个内含两个 int 类型值的数组，而不是指向一个 int 类型值，其声明如下：

```
int (* pz)[2];     // pz 指向一个内含两个 int 类型值的数组
```

以上代码把 pz 声明为指向一个数组的指针，该数组内含两个 int 类型值。为什么要在声明中使用圆括号？因为[]的优先级高于*。考虑下面的声明：

```
int * pax[2];      // pax 是一个内含两个指针元素的数组，每个元素都指向 int 的指针
```

由于[]优先级高，先与 pax 结合，所以 pax 成为一个内含两个元素的数组。然后*表示 pax 数组内含两个指针。最后，int 表示 pax 数组中的指针都指向 int 类型的值。因此，这行代码声明了两个指向 int 的指针。而前面有圆括号的版本，*先与 pz 结合，因此声明的是一个指向数组（内含两个 int 类型的值）的指针。程序清单 10.16 演示了如何使用指向二维数组的指针。

程序清单 10.16　zippo2.c 程序

```
/* zippo2.c --  通过指针获取 zippo 的信息 */
#include <stdio.h>
int main(void)
{
    int zippo[4][2] = { { 2, 4 }, { 6, 8 }, { 1, 3 }, { 5, 7 } };
    int(*pz)[2];
    pz = zippo;

    printf("   pz = %p,    pz + 1 = %p\n",    pz, pz + 1);
    printf("pz[0] = %p, pz[0] + 1 = %p\n",  pz[0], pz[0] + 1);
    printf("  *pz = %p,   *pz + 1 = %p\n",    *pz, *pz + 1);
    printf("pz[0][0] = %d\n", pz[0][0]);
    printf("  *pz[0] = %d\n", *pz[0]);
    printf("    **pz = %d\n", **pz);
    printf("      pz[2][1] = %d\n", pz[2][1]);
    printf("*(*(pz+2) + 1) = %d\n", *(*(pz + 2) + 1));

    return 0;
}
```

下面是该程序的输出：

```
pz = 0x0064fd38,     pz + 1 = 0x0064fd40
pz[0] = 0x0064fd38,  pz[0] + 1 = 0x0064fd3c
  *pz = 0x0064fd38,    *pz + 1 = 0x0064fd3c
pz[0][0] = 2
  *pz[0] = 2
    **pz = 2
      pz[2][1] = 3
*(*(pz+2)  + 1) = 3
```

系统不同，输出的地址可能不同，但是地址之间的关系相同。如前所述，虽然 pz 是一个指针，不是数组名，但是也可以使用 pz[2][1]这样的写法。可以用数组表示法或指针表示法来表示一个数组元素，既

可以使用数组名，也可以使用指针名：

```
zippo[m][n] == *(*(zippo + m) + n)
pz[m][n] == *(*(pz + m) + n)
```

10.7.2 指针的兼容性

指针之间的赋值比数值类型之间的赋值要严格。例如，不用类型转换就可以把 int 类型的值赋给 double 类型的变量，但是两个类型的指针不能这样做。

```
int n = 5;
double x;
int * p1 = &n;
double * pd = &x;
x = n;              // 隐式类型转换
pd = p1;            // 编译时错误
```

更复杂的类型也是如此。假设有如下声明：

```
int * pt;
int (*pa)[3];
int ar1[2][3];
int ar2[3][2];
int **p2; // 一个指向指针的指针
```

有如下的语句：

```
pt = &ar1[0][0];  // 都是指向 int 的指针
pt = ar1[0];      // 都是指向 int 的指针
pt = ar1;         // 无效
pa = ar1;         // 都是指向内含 3 个 int 类型元素数组的指针
pa = ar2;         // 无效
p2 = &pt;         // both pointer-to-int *
*p2 = ar2[0];     // 都是指向 int 的指针
p2 = ar2;         // 无效
```

注意，以上无效的赋值表达式语句中涉及的两个指针都是指向不同的类型。例如，pt 指向一个 int 类型值，而 ar1 指向一个内含 3 和 int 类型元素的数组。类似地，pa 指向一个内含 2 个 int 类型元素的数组，所以它与 ar1 的类型兼容，但是 ar2 指向一个内含 2 个 int 类型元素的数组，所以 pa 与 ar2 不兼容。

上面的最后两个例子有些棘手。变量 p2 是指向指针的指针，它指向的指针指向 int，而 ar2 是指向数组的指针，该数组内含 2 个 int 类型的元素。所以，p2 和 ar2 的类型不同，不能把 ar2 赋给 p2。但是，*p2 是指向 int 的指针，与 ar2[0]兼容。因为 ar2[0]是指向该数组首元素（ar2[0][0]）的指针，所以 ar2[0]也是指向 int 的指针。

一般而言，多重解引用让人费解。例如，考虑下面的代码：

```
int x = 20;
const int y = 23;
int * p1 = &x;
const int * p2 = &y;
const int ** pp2;
p1 = p2;      // 不安全 -- 把 const 指针赋给非 const 指针
p2 = p1;      // 有效 -- 把非 const 指针赋给 const 指针
pp2 = &p1;    // 不安全 -- 嵌套指针类型赋值
```

前面提到过，把 const 指针赋给非 const 指针不安全，因为这样可以使用新的指针改变 const 指针指向的数据。编译器在编译代码时，可能会给出警告，执行这样的代码是未定义的。但是把非 const 指针赋给 const 指针没问题，前提是只进行一级解引用：

```
p2 = p1; // 有效 -- 把非 const 指针赋给 const 指针
```

但是进行两级解引用时，这样的赋值也不安全，例如，考虑下面的代码：

```
const int **pp2;
int *p1;
const int n = 13;
pp2 = &p1;     // 允许，但是这导致 const 限定符失效（根据第 1 行代码，不能通过*pp2 修改它所指向的内容）
*pp2 = &n;     // 有效，两者都声明为 const，但是这将导致 p1 指向 n（*pp2 已被修改）
*p1 = 10; //有效，但是这将改变 n 的值（但是根据第 3 行代码，不能修改 n 的值）
```

发生了什么？如前所示，标准规定了通过非 const 指针更改 const 数据是未定义的。例如，在 Terminal 中（OS X 对底层 UNIX 系统的访问）使用 gcc 编译包含以上代码的小程序，导致 n 最终的值是 13，但是在相同系统下使用 clang 来编译，n 最终的值是 10。两个编译器都给出指针类型不兼容的警告。当然，可以忽略这些警告，但是最好不要相信该程序运行的结果，这些结果都是未定义的。

C const 和 C++ const

C 和 C++中 const 的用法很相似，但是并不完全相同。区别之一是，C++允许在声明数组大小时使用 const 整数，而 C 却不允许。区别之二是，C++的指针赋值检查更严格:

```
const int y;
const int * p2 = &y;
int * p1;
p1 = p2; // C++中不允许这样做，但是C可能只给出警告
```

C++不允许把 const 指针赋给非 const 指针。而 C 则允许这样做，但是如果通过 p1 更改 y，其行为是未定义的。

10.7.3　函数和多维数组

如果要编写处理二维数组的函数，首先要能正确地理解指针才能写出声明函数的形参。在函数体中，通常使用数组表示法进行相关操作。

下面，我们编写一个处理二维数组的函数。一种方法是，利用 for 循环把处理一维数组的函数应用到二维数组的每一行。如下所示：

```
int junk[3][4] = { {2,4,5,8}, {3,5,6,9}, {12,10,8,6} };
int i, j;
int total = 0;
for (i = 0; i < 3 ; i++)
    total += sum(junk[i], 4); // junk[i]是一维数组
```

记住，如果 junk 是二维数组，junk[i]就是一维数组，可将其视为二维数组的一行。这里，sum()函数计算二维数组的每行的总和，然后 for 循环再把每行的总和加起来。

然而，这种方法无法记录行和列的信息。用这种方法计算总和，行和列的信息并不重要。但如果每行代表一年，每列代表一个月，就还需要一个函数计算某列的总和。该函数要知道行和列的信息，可以通过声明正确类型的形参变量来完成，以便函数能正确地传递数组。在这种情况下，数组 junk 是一个内含 3 个数组元素的数组，每个元素是内含 4 个 int 类型值的数组（即 junk 是一个 3 行 4 列的二维数组）。通过

前面的讨论可知，这表明 junk 是一个指向数组（内含 4 个 int 类型值）的指针。可以这样声明函数的形参：

```
void somefunction( int (* pt)[4] );
```

另外，如果当且仅当 pt 是一个函数的形式参数时，可以这样声明：

```
void somefunction( int pt[][4] );
```

注意，第 1 个方括号是空的。空的方括号表明 pt 是一个指针。这样的变量稍后可以用作相同方法作为 junk。下面的程序示例中就是这样做的，如程序清单 10.17 所示。注意该程序清单演示了 3 种等价的原型语法。

程序清单 10.17 array2d.c 程序

```
// array2d.c -- 处理二维数组的函数
#include <stdio.h>
#define ROWS 3
#define COLS 4
void sum_rows(int ar[][COLS], int rows);
void sum_cols(int [][COLS], int);       // 省略形参名，没问题
int sum2d(int(*ar)[COLS], int rows);    // 另一种语法
int main(void)
{
    int junk[ROWS][COLS] = {
            { 2, 4, 6, 8 },
            { 3, 5, 7, 9 },
            { 12, 10, 8, 6 }
    };

    sum_rows(junk, ROWS);
    sum_cols(junk, ROWS);
    printf("Sum of all elements = %d\n", sum2d(junk, ROWS));

    return 0;
}

void sum_rows(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;

    for (r = 0; r < rows; r++)
    {
        tot = 0;
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
        printf("row %d: sum = %d\n", r, tot);
    }
}

void sum_cols(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot;
```

```
        for (c = 0; c < COLS; c++)
        {
            tot = 0;
            for (r = 0; r < rows; r++)
                tot += ar[r][c];
            printf("col %d: sum = %d\n", c, tot);
        }
    }

    int sum2d(int ar[][COLS], int rows)
    {
        int r;
        int c;
        int tot = 0;

        for (r = 0; r < rows; r++)
            for (c = 0; c < COLS; c++)
                tot += ar[r][c];

        return tot;
    }
```

该程序的输出如下：

```
row 0: sum = 20
row 1: sum = 24
row 2: sum = 36
col 0: sum = 17
col 1: sum = 19
col 2: sum = 21
col 3: sum = 23
Sum of all elements = 80
```

程序清单 10.17 中的程序把数组名 junk（即，指向数组首元素的指针，首元素是子数组）和符号常量 ROWS（代表行数 3）作为参数传递给函数。每个函数都把 ar 视为内含数组元素（每个元素是内含 4 个 int 类型值的数组）的数组。列数内置在函数体中，但是行数靠函数传递得到。如果传入函数的行数是 12，那么函数要处理的是 12×4 的数组。因为 rows 是元素的个数，然而，因为每个元素都是数组，或者视为一行，rows 也可以看成是行数。

注意，ar 和 main()中的 junk 都使用数组表示法。因为 ar 和 junk 的类型相同，它们都是指向内含 4 个 int 类型值的数组的指针。

注意，下面的声明不正确：

```
int sum2(int ar[][], int rows);// 错误的声明
```

前面介绍过，编译器会把数组表示法转换成指针表示法。例如，编译器会把 ar[1]转换成 ar+1。编译器对 ar+1 求值，要知道 ar 所指向的对象大小。下面的声明：

```
int sum2(int ar[][4], int rows);    // 有效声明
```

表示 ar 指向一个内含 4 个 int 类型值的数组（在我们的系统中，ar 指向的对象占 16 字节），所以 ar+1 的意思是“该地址加上 16 字节”。如果第 2 对方括号是空的，编译器就不知道该怎样处理。

也可以在第 1 对方括号中写上大小，如下所示，但是编译器会忽略该值：

```
int sum2(int ar[3][4], int rows); // 有效声明，但是 3 将被忽略
```

与使用 typedef（第 5 章和第 14 章中讨论）相比，这种形式方便得多：

```
typedef int arr4[4];                // arr4 是一个内含 4 个 int 的数组
typedef arr4 arr3x4[3];             // arr3x4 是一个内含 3 个 arr4 的数组
int sum2(arr3x4 ar, int rows);      // 与下面的声明相同
int sum2(int ar[3][4], int rows);   // 与下面的声明相同
int sum2(int ar[][4], int rows);    // 标准形式
```

一般而言，声明一个指向 N 维数组的指针时，只能省略最左边方括号中的值：

```
int sum4d(int ar[][12][20][30], int rows);
```

因为第 1 对方括号只用于表明这是一个指针，而其他的方括号则用于描述指针所指向数据对象的类型。下面的声明与该声明等价：

```
int sum4d(int (*ar)[12][20][30], int rows); // ar 是一个指针
```

这里，ar 指向一个 12×20×30 的 int 数组。

10.8　变长数组（VLA）

读者在学习处理二维数组的函数中可能不太理解，为何只把数组的行数作为函数的形参，而列数却内置在函数体内。例如，函数定义如下：

```
#define COLS 4
int sum2d(int ar[][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];
    return tot;
}
```

假设声明了下列数组：

```
int array1[5][4];
int array2[100][4];
int array3[2][4];
```

可以用 sum2d()函数分别计算这些数组的元素之和：

```
tot = sum2d(array1, 5);    // 5×4 数组的元素之和
tot = sum2d(array2, 100);  // 100×4 数组的元素之和
tot = sum2d(array3, 2);    // 2×4 数组的元素之和
```

sum2d()函数之所以能处理这些数组，是因为这些数组的列数固定为 4，而行数被传递给形参 rows，rows 是一个变量。但是如果要计算 6×5 的数组（即 6 行 5 列），就不能使用这个函数，必须重新创建一个 CLOS 为 5 的函数。因为 C 规定，数组的维数必须是常量，不能用变量来代替 COLS。

要创建一个能处理任意大小二维数组的函数，比较繁琐（必须把数组作为一维数组传递，然后让函数计算每行的开始处）。而且，这种方法不好处理 FORTRAN 的子例程，这些子例程都允许在函数调用中指定两个维度。虽然 FORTRAN 是比较老的编程语言，但是在过去的几十年里，数值计算领域的专家已经用 FORTRAN 开发出许多有用的计算库。C 正逐渐替代 FORTRAN，如果能直接转换现有的 FORTRAN 库就好了。

鉴于此，C99 新增了变长数组（*variable-length array*，VLA），允许使用变量表示数组的维度。如下所示：

```
int quarters = 4;
int regions = 5;
double sales[regions][quarters];    // 一个变长数组（VLA）
```

前面提到过，变长数组有一些限制。变长数组必须是自动存储类别，这意味着无论在函数中声明还是作为函数形参声明，都不能使用 static 或 extern 存储类别说明符（第 12 章介绍）。而且，不能在声明中初始化它们。最终，C11 把变长数组作为一个可选特性，而不是必须强制实现的特性。

注意　变长数组不能改变大小

变长数组中的“变”不是指可以修改已创建数组的大小。一旦创建了变长数组，它的大小则保持不变。这里的“变”指的是：在创建数组时，可以使用变量指定数组的维度。

由于变长数组是 C 语言的新特性，目前完全支持这一特性的编译器不多。下面我们来看一个简单的例子：如何编写一个函数，计算 int 的二维数组所有元素之和。

首先，要声明一个带二维变长数组参数的函数，如下所示：

```
int sum2d(int rows, int cols, int ar[rows][cols]); // ar 是一个变长数组（VLA）
```

注意前两个形参（rows 和 cols）用作第 3 个形参二维数组 ar 的两个维度。因为 ar 的声明要使用 rows 和 cols，所以在形参列表中必须在声明 ar 之前先声明这两个形参。因此，下面的原型是错误的：

```
int sum2d(int ar[rows][cols], int rows, int cols); // 无效的顺序
```

C99/C11 标准规定，可以省略原型中的形参名，但是在这种情况下，必须用星号来代替省略的维度：

```
int sum2d(int, int, int ar[*][*]); // ar 是一个变长数组（VLA），省略了维度形参名
```

其次，该函数的定义如下：

```
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];
    return tot;
}
```

该函数除函数头与传统的 C 函数（程序清单 10.17）不同外，还把符号常量 COLS 替换成变量 cols。这是因为在函数头中使用了变长数组。由于用变量代表行数和列数，所以新的 sum2d()现在可以处理任意大小的二维 int 数组，如程序清单 10.18 所示。但是，该程序要求编译器支持变长数组特性。另外，该程序还演示了以变长数组作为形参的函数既可处理传统 C 数组，也可处理变长数组。

程序清单 10.18　vararr2d.c 程序

```
//vararr2d.c -- 使用变长数组的函数
#include <stdio.h>
#define ROWS 3
#define COLS 4
int sum2d(int rows, int cols, int ar[rows][cols]);
int main(void)
{
    int i, j;
```

```
    int rs = 3;
    int cs = 10;
    int junk[ROWS][COLS] = {
            { 2, 4, 6, 8 },
            { 3, 5, 7, 9 },
            { 12, 10, 8, 6 }
    };

    int morejunk[ROWS - 1][COLS + 2] = {
            { 20, 30, 40, 50, 60, 70 },
            { 5, 6, 7, 8, 9, 10 }
    };

    int varr[rs][cs];  // 变长数组（VLA）

    for (i = 0; i < rs; i++)
        for (j = 0; j < cs; j++)
            varr[i][j] = i * j + j;

    printf("3x5 array\n");
    printf("Sum of all elements = %d\n", sum2d(ROWS, COLS, junk));

    printf("2x6 array\n");
    printf("Sum of all elements = %d\n", sum2d(ROWS - 1, COLS + 2, morejunk));

    printf("3x10 VLA\n");
    printf("Sum of all elements = %d\n", sum2d(rs, cs, varr));

    return 0;
}

// 带变长数组形参的函数
int sum2d(int rows, int cols, int ar[rows][cols])
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < cols; c++)
            tot += ar[r][c];

    return tot;
}
```

下面是该程序的输出：

```
3x5 array
Sum of all elements = 80
2x6 array
Sum of all elements = 315
3x10 VLA
Sum of all elements = 270
```

需要注意的是，在函数定义的形参列表中声明的变长数组并未实际创建数组。和传统的语法类似，变长数组名实际上是一个指针。这说明带变长数组形参的函数实际上是在原始数组中处理数组，因此可以修

改传入的数组。下面的代码段指出指针和实际数组是何时声明的：

```
        int thing[10][6];
        twoset(10,6,thing);
        ...
    }

    void twoset (int n, int m, int ar[n][m]) // ar 是一个指向数组（内含 m 个 int 类型的值）的指针
    {
        int temp[n][m];    // temp 是一个 n×m 的 int 数组
        temp[0][0] = 2;    // 设置 temp 的一个元素为 2
        ar[0][0] = 2;      // 设置 thing[0][0]为 2
    }
```

如上代码所示调用 twoset()时，ar 成为指向 thing[0]的指针，temp 被创建为 10×6 的数组。因为 ar 和 thing 都是指向 thing[0]的指针，ar[0][0]与 thing[0][0]访问的数据位置相同。

const 和数组大小

是否可以在声明数组时使用 const 变量？

```
const int SZ = 80;
...
double ar[SZ]; // 是否允许？
```

C90 标准不允许（也可能允许）。数组的大小必须是给定的整型常量表达式，可以是整型常量组合，如 20、sizeof 表达式或其他不是 const 的内容。由于 C 实现可以扩大整型常量表达式的范围，所以可能会允许使用 const，但是这种代码可能无法移植。

C99/C11 标准允许在声明变长数组时使用 const 变量。所以该数组的定义必须是声明在块中的自动存储类别数组。

变长数组还允许动态内存分配，这说明可以在程序运行时指定数组的大小。普通 C 数组都是静态内存分配，即在编译时确定数组的大小。由于数组大小是常量，所以编译器在编译时就知道了。第 12 章将详细介绍动态内存分配。

10.9　复合字面量

假设给带 int 类型形参的函数传递一个值，要传递 int 类型的变量，但是也可以传递 int 类型常量，如 5。在 C99 标准以前，对于带数组形参的函数，情况不同，可以传递数组，但是没有等价的数组常量。C99 新增了复合字面量（*compound literal*）。字面量是除符号常量外的常量。例如，5 是 int 类型字面量，81.3 是 double 类型的字面量，'Y'是 char 类型的字面量，"elephant"是字符串字面量。发布 C99 标准的委员会认为，如果有代表数组和结构内容的复合字面量，在编程时会更方便。

对于数组，复合字面量类似数组初始化列表，前面是用括号括起来的类型名。例如，下面是一个普通的数组声明：

```
int diva[2] = {10, 20};
```

下面的复合字面量创建了一个和 diva 数组相同的匿名数组，也有两个 int 类型的值：

```
(int [2]){10, 20}      // 复合字面量
```

注意，去掉声明中的数组名，留下的 int [2]即是复合字面量的类型名。

初始化有数组名的数组时可以省略数组大小，复合字面量也可以省略大小，编译器会自动计算数组当前的元素个数：

```
(int []){50, 20, 90} // 内含 3 个元素的复合字面量
```

因为复合字面量是匿名的，所以不能先创建然后再使用它，必须在创建的同时使用它。使用指针记录地址就是一种用法。也就是说，可以这样用：

```
int * pt1;
pt1 = (int [2]) {10, 20};
```

注意，该复合字面量的字面常量与上面创建的 diva 数组的字面常量完全相同。与有数组名的数组类似，复合字面量的类型名也代表首元素的地址，所以可以把它赋给指向 int 的指针。然后便可使用这个指针。例如，本例中*pt1 是 10，pt1[1]是 20。

还可以把复合字面量作为实际参数传递给带有匹配形式参数的函数：

```
int sum(const int ar[], int n);
...
int total3;
total3 = sum((int []){4,4,4,5,5,5}, 6);
```

这里，第 1 个实参是内含 6 个 int 类型值的数组，和数组名类似，这同时也是该数组首元素的地址。这种用法的好处是，把信息传入函数前不必先创建数组，这是复合字面量的典型用法。

可以把这种用法应用于二维数组或多维数组。例如，下面的代码演示了如何创建二维 int 数组并储存其地址：

```
int (*pt2)[4];     // 声明一个指向二维数组的指针，该数组内含 2 个数组元素，
                   // 每个元素是内含 4 个 int 类型值的数组
pt2 = (int [2][4]) { {1,2,3,-9}, {4,5,6,-8} };
```

如上所示，该复合字面量的类型是 int [2][4]，即一个 2×4 的 int 数组。

程序清单 10.19 把上述例子放进一个完整的程序中。

程序清单 10.19 flc.c 程序

```
// flc.c -- 有趣的常量
#include <stdio.h>
#define COLS 4
int sum2d(const int ar[][COLS], int rows);
int sum(const int ar[], int n);
int main(void)
{
    int total1, total2, total3;
    int * pt1;
    int(*pt2)[COLS];

    pt1 = (int[2]) { 10, 20 };
    pt2 = (int[2][COLS]) { {1, 2, 3, -9}, { 4, 5, 6, -8 } };

    total1 = sum(pt1, 2);
    total2 = sum2d(pt2, 2);
    total3 = sum((int []){ 4, 4, 4, 5, 5, 5 }, 6);
    printf("total1 = %d\n", total1);
    printf("total2 = %d\n", total2);
    printf("total3 = %d\n", total3);

    return 0;
```

```
}

int sum(const int ar [], int n)
{
    int i;
    int total = 0;

    for (i = 0; i < n; i++)
        total += ar[i];

    return total;
}

int sum2d(const int ar [][COLS], int rows)
{
    int r;
    int c;
    int tot = 0;

    for (r = 0; r < rows; r++)
        for (c = 0; c < COLS; c++)
            tot += ar[r][c];

    return tot;
}
```

要支持 C99 的编译器才能正常运行该程序示例（目前并不是所有的编译器都支持），其输出如下：

```
total1 = 30
total2 = 4
total3 = 27
```

记住，复合字面量是提供只临时需要的值的一种手段。复合字面量具有块作用域（第 12 章将介绍相关内容），这意味着一旦离开定义复合字面量的块，程序将无法保证该字面量是否存在。也就是说，复合字面量的定义在最内层的花括号中。

10.10　关键概念

数组用于储存相同类型的数据。C 把数组看作是*派生类型*，因为数组是建立在其他类型的基础上。也就是说，无法简单地声明一个数组。在声明数组时必须说明其元素的类型，如 `int` 类型的数组、`float` 类型的数组，或其他类型的数组。所谓的其他类型也可以是数组类型，这种情况下，创建的是数组的数组（或称为二维数组）。

通常编写一个函数来处理数组，这样在特定的函数中解决特定的问题，有助于实现程序的模块化。在把数组名作为实际参数时，传递给函数的不是整个数组，而是数组的地址（因此，函数对应的形式参数是指针）。为了处理数组，函数必须知道从何处开始读取数据和要处理多少个数组元素。数组地址提供了“地址”，“元素个数”可以内置在函数中或作为单独的参数传递。第 2 种方法更普遍，因为这样做可以让同一个函数处理不同大小的数组。

数组和指针的关系密切，同一个操作可以用数组表示法或指针表示法。它们之间的关系允许你在处理数组的函数中使用数组表示法，即使函数的形式参数是一个指针，而不是数组。

对于传统的 C 数组，必须用常量表达式指明数组的大小，所以数组大小在编译时就已确定。C99/C11

新增了变长数组，可以用变量表示数组大小。这意味着变长数组的大小延迟到程序运行时才确定。

10.11　本章小结

数组是一组数据类型相同的元素。数组元素按顺序储存在内存中，通过整数下标（或索引）可以访问各元素。在 C 中，数组首元素的下标是 0，所以对于内含 n 个元素的数组，其最后一个元素的下标是 n-1。作为程序员，要确保使用有效的数组下标，因为编译器和运行的程序都不会检查下标的有效性。

声明一个简单的一维数组形式如下：

```
type name [ size ];
```

这里，type 是数组中每个元素的数据类型，name 是数组名，size 是数组元素的个数。对于传统的 C 数组，要求 size 是整型常量表达式。但是 C99/C11 允许使用整型非常量表达式。这种情况下的数组被称为变长数组。

C 把数组名解释为该数组首元素的地址。换言之，数组名与指向该数组首元素的指针等价。概括地说，数组和指针的关系十分密切。如果 ar 是一个数组，那么表达式 ar[i]和*(ar+i)等价。

对于 C 语言而言，不能把整个数组作为参数传递给函数，但是可以传递数组的地址。然后函数可以使用传入的地址操控原始数组。如果函数没有修改原始数组的意图，应在声明函数的形式参数时使用关键字 const。在被调函数中可以使用数组表示法或指针表示法，无论用哪种表示法，实际上使用的都是指针变量。

指针加上一个整数或递增指针，指针的值以所指向对象的大小为单位改变。也就是说，如果 pd 指向一个数组的 8 字节 double 类型值，那么 pd 加 1 意味着其值加 8，以便它指向该数组的下一个元素。

二维数组即是数组的数组。例如，下面声明了一个二维数组：

```
double sales[5][12];
```

该数组名为 sales，有 5 个元素（一维数组），每个元素都是一个内含 12 个 double 类型值的数组。第 1 个一维数组是 sales[0]，第 2 个一维数组是 sales[1]，以此类推，每个元素都是内含 12 个 double 类型值的数组。使用第 2 个下标可以访问这些一维数组中的特定元素。例如，sales[2][5]是 slaes[2]的第 6 个元素，而 sales[2]是 sales 的第 3 个元素。

C 语言传递多维数组的传统方法是把数组名（即数组的地址）传递给类型匹配的指针形参。声明这样的指针形参要指定所有的数组维度，除了第 1 个维度。传递的第 1 个维度通常作为第 2 个参数。例如，为了处理前面声明的 sales 数组，函数原型和函数调用如下：

```
void display(double ar[][12], int rows);
...
display(sales, 5);
```

变长数组提供第 2 种语法，把数组维度作为参数传递。在这种情况下，对应函数原型和函数调用如下：

```
void display(int rows, int cols, double ar[rows][cols]);
...
display(5, 12, sales);
```

虽然上述讨论中使用的是 int 类型的数组和 double 类型的数组，其他类型的数组也是如此。然而，字符串有一些特殊的规则，这是由于其末尾的空字符所致。有了这个空字符，不用传递数组的大小，函数通过检测字符串的末尾也知道在何处停止。我们将在第 11 章中详细介绍。

10.12　复习题

复习题的参考答案在附录 A 中。

1. 下面的程序将打印什么内容？

```
#include <stdio.h>
int main(void)
{
    int ref[] = { 8, 4, 0, 2 };
    int *ptr;
    int index;

    for (index = 0, ptr = ref; index < 4; index++, ptr++)
        printf("%d %d\n", ref[index], *ptr);
    return 0;
}
```

2. 在复习题 1 中，ref 有多少个元素？

3. 在复习题 1 中，ref 的地址是什么？ref + 1 是什么意思？++ref 指向什么？

4. 在下面的代码中，*ptr 和*(ptr + 2)的值分别是什么？

   a.

```
int *ptr;
int torf[2][2] = {12, 14, 16};
ptr = torf[0];
```

   b.

```
int * ptr;
int fort[2][2] = { {12}, {14,16} };
ptr = fort[0];
```

5. 在下面的代码中，**ptr 和**(ptr + 1)的值分别是什么？

   a.

```
int (*ptr)[2];
int torf[2][2] = {12, 14, 16};
ptr = torf;
```

   b.

```
int (*ptr)[2];
int fort[2][2] = { {12}, {14,16} };
ptr = fort;
```

6. 假设有下面的声明：

```
int grid[30][100];
```

   a. 用 1 种写法表示 grid[22][56]

   b. 用 2 种写法表示 grid[22][0]

   c. 用 3 种写法表示 grid[0][0]

7. 正确声明以下各变量：

   a. digits 是一个内含 10 个 int 类型值的数组

   b. rates 是一个内含 6 个 float 类型值的数组

c．mat 是一个内含 3 个元素的数组，每个元素都是内含 5 个整数的数组

d．psa 是一个内含 20 个元素的数组，每个元素都是指向 int 的指针

e．pstr 是一个指向数组的指针，该数组内含 20 个 char 类型的值

8．

a．声明一个内含 6 个 int 类型值的数组，并初始化各元素为 1、2、4、8、16、32

b．用数组表示法表示 a 声明的数组的第 3 个元素（其值为 4）

c．假设编译器支持 C99/C11 标准，声明一个内含 100 个 int 类型值的数组，并初始化最后一个元素为-1，其他元素不考虑

d．假设编译器支持 C99/C11 标准，声明一个内含 100 个 int 类型值的数组，并初始化下标为 5、10、11、12、3 的元素为 101，其他元素不考虑

9．内含 10 个元素的数组下标范围是什么？

10．假设有下面的声明：

```
float rootbeer[10], things[10][5], *pf, value = 2.2;
int i = 3;
```

判断以下各项是否有效：

a. `rootbeer[2] = value;`
b. `scanf("%f", &rootbeer );`
c. `rootbeer = value;`
d. `printf("%f", rootbeer);`
e. `things[4][4] = rootbeer[3];`
f. `things[5] = rootbeer;`
g. `pf = value;`
h. `pf = rootbeer;`

11．声明一个 800×600 的 int 类型数组。

12．下面声明了 3 个数组：

```
double trots[20];
short clops[10][30];
long shots[5][10][15];
```

a．分别以传统方式和以变长数组为参数的方式编写处理 trots 数组的 void 函数原型和函数调用

b．分别以传统方式和以变长数组为参数的方式编写处理 clops 数组的 void 函数原型和函数调用

c．分别以传统方式和以变长数组为参数的方式编写处理 shots 数组的 void 函数原型和函数调用

13．下面有两个函数原型：

```
void show(const double ar[], int n);         // n 是数组元素的个数
void show2(const double ar2[][3], int n);    // n 是二维数组的行数
```

a．编写一个函数调用，把一个内含 8、3、9 和 2 的复合字面量传递给 show()函数。

b．编写一个函数调用，把一个 2 行 3 列的复合字面量（8、3、9 作为第 1 行，5、4、1 作为第 2 行）传递给 show2()函数。

10.13 编程练习

1．修改程序清单 10.7 的 rain.c 程序，用指针进行计算（仍然要声明并初始化数组）。

2. 编写一个程序，初始化一个 double 类型的数组，然后把该数组的内容拷贝至 3 个其他数组中（在 main()中声明这 4 个数组）。使用带数组表示法的函数进行第 1 份拷贝。使用带指针表示法和指针递增的函数进行第 2 份拷贝。把目标数组名、源数组名和待拷贝的元素个数作为前两个函数的参数。第 3 个函数以目标数组名、源数组名和指向源数组最后一个元素后面的元素的指针。也就是说，给定以下声明，则函数调用如下所示：

```
double source[5] = {1.1, 2.2, 3.3, 4.4, 5.5};
double target1[5];
double target2[5];
double target3[5];
copy_arr(target1, source, 5);
copy_ptr(target2, source, 5);

copy_ptrs(target3, source, source + 5);
```

3. 编写一个函数，返回储存在 int 类型数组中的最大值，并在一个简单的程序中测试该函数。

4. 编写一个函数，返回储存在 double 类型数组中最大值的下标，并在一个简单的程序中测试该函数。

5. 编写一个函数，返回储存在 double 类型数组中最大值和最小值的差值，并在一个简单的程序中测试该函数。

6. 编写一个函数，把 double 类型数组中的数据倒序排列，并在一个简单的程序中测试该函数。

7. 编写一个程序，初始化一个 double 类型的二维数组，使用编程练习 2 中的一个拷贝函数把该数组中的数据拷贝至另一个二维数组中（因为二维数组是数组的数组，所以可以使用处理一维数组的拷贝函数来处理数组中的每个子数组）。

8. 使用编程练习 2 中的拷贝函数，把一个内含 7 个元素的数组中第 3～第 5 个元素拷贝至内含 3 个元素的数组中。该函数本身不需要修改，只需要选择合适的实际参数（实际参数不需要是数组名和数组大小，只需要是数组元素的地址和待处理元素的个数）。

9. 编写一个程序，初始化一个 double 类型的 3×5 二维数组，使用一个处理变长数组的函数将其拷贝至另一个二维数组中。还要编写一个以变长数组为形参的函数以显示两个数组的内容。这两个函数应该能处理任意 N×M 数组（如果编译器不支持变长数组，就使用传统 C 函数处理 N×5 的数组）。

10. 编写一个函数，把两个数组中相对应的元素相加，然后把结果储存到第 3 个数组中。也就是说，如果数组 1 中包含的值是 2、4、5、8，数组 2 中包含的值是 1、0、4、6，那么该函数把 3、4、9、14 赋给第 3 个数组。函数接受 3 个数组名和一个数组大小。在一个简单的程序中测试该函数。

11. 编写一个程序，声明一个 int 类型的 3×5 二维数组，并用合适的值初始化它。该程序打印数组中的值，然后各值翻倍（即是原值的 2 倍），并显示出各元素的新值。编写一个函数显示数组的内容，再编写一个函数把各元素的值翻倍。这两个函数都以函数名和行数作为参数。

12. 重写程序清单 10.7 的 rain.c 程序，把 main()中的主要任务都改成用函数来完成。

13. 编写一个程序，提示用户输入 3 组数，每组数包含 5 个 double 类型的数（假设用户都正确地响应，不会输入非数值数据）。该程序应完成下列任务。

   a. 把用户输入的数据储存在 3×5 的数组中

   b. 计算每组（5 个）数据的平均值

   c. 计算所有数据的平均值

   d. 找出这 15 个数据中的最大值

e．打印结果

每个任务都要用单独的函数来完成（使用传统 C 处理数组的方式）。完成任务 b，要编写一个计算并返回一维数组平均值的函数，利用循环调用该函数 3 次。对于处理其他任务的函数，应该把整个数组作为参数，完成任务 c 和 d 的函数应把结果返回主调函数。

14．以变长数组作为函数形参，完成编程练习 13。

第 11 章

字符串和字符串函数

本章介绍以下内容：

- 函数：gets()、gets_s()、fgets()、puts()、fputs()、strcat()、strncat()、strcmp()、strncmp()、strcpy()、strncpy()、sprintf()、strchr()
- 创建并使用字符串
- 使用 C 库中的字符和字符串函数，并创建自定义的字符串函数
- 使用命令行参数

字符串是 C 语言中最有用、最重要的数据类型之一。虽然我们一直在使用字符串，但是要学的东西还很多。C 库提供大量的函数用于读写字符串、拷贝字符串、比较字符串、合并字符串、查找字符串等。通过本章的学习，读者将进一步提高自己的编程水平。

11.1 表示字符串和字符串 I/O

第 4 章介绍过，字符串是以空字符（\0）结尾的 char 类型数组。因此，可以把上一章学到的数组和指针的知识应用于字符串。不过，由于字符串十分常用，所以 C 提供了许多专门用于处理字符串的函数。本章将讨论字符串的性质、如何声明并初始化字符串、如何在程序中输入和输出字符串，以及如何操控字符串。

程序清单 11.1 演示了在程序中表示字符串的几种方式。

程序清单 11.1 strings1.c **程序**

```
//  strings1.c
#include <stdio.h>
#define MSG "I am a symbolic string constant."
#define MAXLENGTH 81
int main(void)
{
    char words[MAXLENGTH] = "I am a string in an array.";
    const char * pt1 = "Something is pointing at me.";
    puts("Here are some strings:");
    puts(MSG);
    puts(words);
    puts(pt1);
    words[8] = 'p';
    puts(words);

    return 0;
}
```

和 printf()函数一样，puts()函数也属于 stdio.h 系列的输入/输出函数。但是，与 printf()

不同的是，puts()函数只显示字符串，而且自动在显示的字符串末尾加上换行符。下面是该程序的输出：

```
Here are some strings:
I am an old-fashioned symbolic string constant.
I am a string in an array.
Something is pointing at me.
I am a spring in an array.
```

我们先分析一下该程序中定义字符串的几种方法，然后再讲解把字符串读入程序涉及的一些操作，最后学习如何输出字符串。

11.1.1　在程序中定义字符串

程序清单 11.1 中使用了多种方法（即字符串常量、char 类型数组、指向 char 的指针）定义字符串。程序应该确保有足够的空间储存字符串，这一点我们稍后讨论。

1. 字符串字面量（字符串常量）

用双引号括起来的内容称为*字符串字面量*（*string literal*），也叫作*字符串常量*（*string constant*）。双引号中的字符和编译器自动加入末尾的\0字符，都作为字符串储存在内存中，所以"I am a symbolic stringconstant."、"I am a string in an array."、"Something is pointed at me."、"Here are some strings:"都是字符串字面量。

从 ANSI C 标准起，如果字符串字面量之间没有间隔，或者用空白字符分隔，C 会将其视为串联起来的字符串字面量。例如：

```
char greeting[50] = "Hello, and"" how are" " you"
                    " today!";
```

与下面的代码等价：

```
char greeting[50] = "Hello, and how are you today!";
```

如果要在字符串内部使用双引号，必须在双引号前面加上一个反斜杠（\）：

```
printf("\"Run, Spot, run!\" exclaimed Dick.\n");
```

输出如下：

```
"Run, Spot, run!" exclaimed Dick.
```

字符串常量属于*静态存储类别*（*static storage class*），这说明如果在函数中使用字符串常量，该字符串只会被储存一次，在整个程序的生命期内存在，即使函数被调用多次。用双引号括起来的内容被视为指向该字符串储存位置的指针。这类似于把数组名作为指向该数组位置的指针。如果确实如此，程序清单 11.2 中的程序会输出什么？

程序清单 11.2　strptr.c 程序

```
/* strptr.c -- 把字符串看作指针 */
#include <stdio.h>
int main(void)
{
    printf("%s, %p, %c\n", "We", "are", *"space farers");

    return 0;
}
```

printf()根据%s 转换说明打印 We，根据%p 转换说明打印一个地址。因此，如果"are"代表一个地址，printf()将打印该字符串首字符的地址（如果使用 ANSI 之前的实现，可能要用%u 或%lu 代替%p）。

最后，*"space farers"表示该字符串所指向地址上储存的值，应该是字符串*"space farers"的首字符。是否真的是这样？下面是该程序的输出：

```
We, 0x100000f61, s
```

2. 字符串数组和初始化

定义字符串数组时，必须让编译器知道需要多少空间。一种方法是用足够空间的数组储存字符串。在下面的声明中，用指定的字符串初始化数组 m1：

```
const char m1[40] = "Limit yourself to one line's worth.";
```

const 表明不会更改这个字符串。

这种形式的初始化比标准的数组初始化形式简单得多：

```
const char m1[40] = { 'L','i', 'm', 'i', 't', ' ', 'y', 'o', 'u', 'r', 's', 'e', 'l',
                      'f', ' ', 't', 'o', ' ', 'o', 'n', 'e', ' ','l', 'i', 'n', 'e',
                      '\", 's', ' ', 'w', 'o', 'r','t', 'h', '.', '\0'
};
```

注意最后的空字符。没有这个空字符，这就不是一个字符串，而是一个字符数组。

在指定数组大小时，要确保数组的元素个数至少比字符串长度多 1（为了容纳空字符）。所有未被使用的元素都被自动初始化为 0（这里的 0 指的是 char 形式的空字符，不是数字字符 0），如图 11.1 所示。

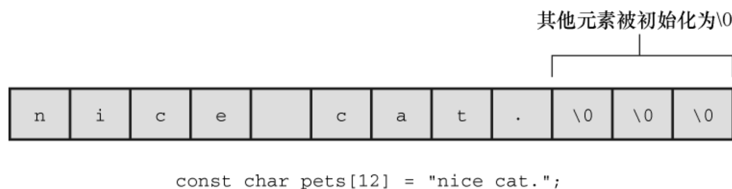


图 11.1　初始化数组

通常，让编译器确定数组的大小很方便。回忆一下，省略数组初始化声明中的大小，编译器会自动计算数组的大小：

```
const char m2[] = "If you can't think of anything, fake it.";
```

让编译器确定初始化字符数组的大小很合理。因为处理字符串的函数通常都不知道数组的大小，这些函数通过查找字符串末尾的空字符确定字符串在何处结束。

让编译器计算数组的大小只能用在初始化数组时。如果创建一个稍后再填充的数组，就必须在声明时指定大小。声明数组时，数组大小必须是可求值的整数。在 C99 新增变长数组之前，数组的大小必须是整型常量，包括由整型常量组成的表达式。

```
int n = 8;
char cookies[1];        // 有效
char cakes[2 + 5];// 有效，数组大小是整型常量表达式
char pies[2*sizeof(long double) + 1]; // 有效
char crumbs[n];         // 在 C99 标准之前无效，C99 标准之后这种数组是变长数组
```

字符数组名和其他数组名一样，是该数组首元素的地址。因此，假设有下面的初始化：

```
char car[10] = "Tata";
```

那么，以下表达式都为真：

car == &car[0]、*car == 'T'、*(car+1) == car[1] == 'a'。

还可以使用指针表示法创建字符串。例如，程序清单 11.1 中使用了下面的声明：

```
const char * pt1 = "Something is pointing at me.";
```

该声明和下面的声明几乎相同：

```
const char ar1[] = "Something is pointing at me.";
```

以上两个声明表明，pt1 和 ar1 都是该字符串的地址。在这两种情况下，带双引号的字符串本身决定了预留给字符串的存储空间。尽管如此，这两种形式并不完全相同。

3. 数组和指针

数组形式和指针形式有何不同？以上面的声明为例，数组形式（ar1[]）在计算机的内存中分配为一个内含 29 个元素的数组（每个元素对应一个字符，还加上一个末尾的空字符'\0'），每个元素被初始化为字符串字面量对应的字符。通常，字符串都作为可执行文件的一部分储存在数据段中。当把程序载入内存时，也载入了程序中的字符串。字符串储存在*静态存储区*（*static memory*）中。但是，程序在开始运行时才会为该数组分配内存。此时，才将字符串拷贝到数组中（第 12 章将详细讲解）。注意，此时字符串有两个副本。一个是在静态内存中的字符串字面量，另一个是储存在 ar1 数组中的字符串。

此后，编译器便把数组名 ar1 识别为该数组首元素地址（&ar1[0]）的别名。这里关键要理解，在数组形式中，ar1 是*地址常量*。不能更改 ar1，如果改变了 ar1，则意味着改变了数组的存储位置（即地址）。可以进行类似 ar1+1 这样的操作，标识数组的下一个元素。但是不允许进行++ar1 这样的操作。递增运算符只能用于变量名前（或概括地说，只能用于可修改的左值），不能用于常量。

指针形式（*pt1）也使得编译器为字符串在静态存储区预留 29 个元素的空间。另外，一旦开始执行程序，它会为*指针变量* pt1 留出一个储存位置，并把字符串的地址储存在指针变量中。该变量最初指向该字符串的首字符，但是它的值可以改变。因此，可以使用递增运算符。例如，++pt1 将指向第 2 个字符（o）。

字符串字面量被视为 const 数据。由于 pt1 指向这个 const 数据，所以应该把 pt1 声明为指向 const 数据的指针。这意味着不能用 pt1 改变它所指向的数据，但是仍然可以改变 pt1 的值（即，pt1 指向的位置）。如果把一个字符串字面量拷贝给一个数组，就可以随意改变数据，除非把数组声明为 const。

总之，初始化数组把静态存储区的字符串拷贝到数组中，而初始化指针只把字符串的地址拷贝给指针。程序清单 11.3 演示了这一点。

程序清单 11.3　addresses.c 程序

```
// addresses.c -- 字符串的地址
#define MSG "I'm special"

#include <stdio.h>
int main()
{
    char ar[] = MSG;
    const char *pt = MSG;
    printf("address of \"I'm special\": %p \n", "I'm special");
    printf("             address ar: %p\n", ar);
    printf("             address pt: %p\n", pt);
    printf("         address of MSG: %p\n", MSG);
    printf("address of \"I'm special\": %p \n", "I'm special");

    return 0;
}
```

下面是在我们的系统中运行该程序后的输出：

```
address of "I'm special": 0x100000f10
              address ar: 0x7fff5fbff858
              address pt: 0x100000f10
          address of MSG: 0x100000f10
address of "I'm special": 0x100000f10
```

该程序的输出说明了什么？第一，pt 和 MSG 的地址相同，而 ar 的地址不同，这与我们前面讨论的内容一致。第二，虽然字符串字面量"I'm special"在程序的两个 printf()函数中出现了两次，但是编译器只使用了一个存储位置，而且与 MSG 的地址相同。编译器可以把多次使用的相同字面量储存在一处或多处。另一个编译器可能在不同的位置储存 3 个"I'm special"。第三，静态数据使用的内存与 ar 使用的动态内存不同。不仅值不同，特定编译器甚至使用不同的位数表示两种内存。

数组和指针表示字符串的区别是否很重要？通常不太重要，但是这取决于想用程序做什么。我们来进一步讨论这个主题。

4. 数组和指针的区别

初始化字符数组来储存字符串和初始化指针来指向字符串有何区别（“指向字符串”的意思是指向字符串的首字符）？例如，假设有下面两个声明：

```
char heart[] = "I love Tillie!";
const char *head = "I love Millie!";
```

两者主要的区别是：数组名 heart 是常量，而指针名 head 是变量。那么，实际使用有什么区别？

首先，两者都可以使用数组表示法：

```
for (i = 0; i < 6; i++)
    putchar(heart[i]);
putchar('\n');
for (i = 0; i < 6; i++)
    putchar(head[i]);
putchar('\n');
```

上面两段代码的输出是：

```
I love
I love
```

其次，两者都能进行指针加法操作：

```
for (i = 0; i < 6; i++)
    putchar(*(heart + i));
putchar('\n');
for (i = 0; i < 6; i++)
    putchar(*(head + i));
putchar('\n');
```

输出如下：

```
I love
I love
```

但是，只有指针表示法可以进行递增操作：

```
while (*(head) != '\0')    /* 在字符串末尾处停止*/
    putchar(*(head++));    /* 打印字符，指针指向下一个位置 */
```

这段代码的输出如下：

```
I love Millie!
```

假设想让 head 和 heart 统一，可以这样做：

```
head = heart;      /* head 现在指向数组 heart */
```

这使得 head 指针指向 heart 数组的首元素。

但是，不能这样做：

```
heart = head;      /* 非法构造，不能这样写 */
```

这类似于 x = 3;和 3 = x;的情况。赋值运算符的左侧必须是变量（或概括地说是可修改的左值），如*pt_int。顺带一提，head = heart;不会导致 head 指向的字符串消失，这样做只是改变了储存在 head 中的地址。除非已经保存了"I love Millie!"的地址，否则当 head 指向别处时，就无法再访问该字符串。

另外，还可以改变 heart 数组中元素的信息：

```
heart[7]= 'M';或者*(heart + 7) = 'M';
```

数组的元素是变量（除非数组被声明为 const），但是数组名不是变量。

我们来看一下未使用 const 限定符的指针初始化：

```
char * word = "frame";
```

是否能使用该指针修改这个字符串？

```
word[1] = 'l'; // 是否允许?
```

编译器可能允许这样做，但是对当前的 C 标准而言，这样的行为是未定义的。例如，这样的语句可能导致内存访问错误。原因前面提到过，编译器可以使用内存中的一个副本来表示所有完全相同的字符串字面量。例如，下面的语句都引用字符串"Klingon"的一个内存位置：

```
char * p1 = "Klingon";
p1[0] = 'F'; // ok?
printf("Klingon");
printf(": Beware the %ss!\n", "Klingon");
```

也就是说，编译器可以用相同的地址替换每个"Klingon"实例。如果编译器使用这种单次副本表示法，并允许 p1[0]修改'F'，那将影响所有使用该字符串的代码。所以以上语句打印字符串字面量"Klingon"时实际上显示的是"Flingon"：

```
Flingon: Beware the Flingons!
```

实际上在过去，一些编译器由于这方面的原因，其行为难以捉摸，而另一些编译器则导致程序异常中断。因此，建议在把指针初始化为字符串字面量时使用 const 限定符：

```
const char * pl = "Klingon";    // 推荐用法
```

然而，把非 const 数组初始化为字符串字面量却不会导致类似的问题。因为数组获得的是原始字符串的副本。

总之，如果不修改字符串，不要用指针指向字符串字面量。

5. 字符串数组

如果创建一个字符数组会很方便，可以通过数组下标访问多个不同的字符串。程序清单 11.4 演示了两种方法：指向字符串的指针数组和 char 类型数组的数组。

程序清单 11.4　arrchar.c 程序

```
//  arrchar.c -- 指针数组，字符串数组
#include <stdio.h>
#define SLEN 40
#define LIM 5
int main(void)
{
```

```
    const char *mytalents[LIM] = {
        "Adding numbers swiftly",
        "Multiplying accurately", "Stashing data",
        "Following instructions to the letter",
        "Understanding the C language"
    };
    char yourtalents[LIM][SLEN] = {
        "Walking in a straight line",
        "Sleeping", "Watching television",
        "Mailing letters", "Reading email"
    };
    int i;

    puts("Let's compare talents.");
    printf("%-36s  %-25s\n", "My Talents", "Your Talents");
    for (i = 0; i < LIM; i++)
        printf("%-36s  %-25s\n", mytalents[i], yourtalents[i]);
    printf("\nsizeof mytalents: %zd, sizeof yourtalents: %zd\n",
                   sizeof(mytalents), sizeof(yourtalents));

    return 0;
}
```

下面是该程序的输出：

```
Let's compare talents.
My Talents                            Your Talents
Adding numbers swiftly                Walking in a straight line
Multiplying accurately                Sleeping
Stashing data                         Watching television
Following instructions to the letter  Mailing letters
Understanding the C language          Reading email

sizeof mytalents: 40, sizeof yourtalents: 200
```

从某些方面来看，mytalents 和 yourtalents 非常相似。两者都代表 5 个字符串。使用一个下标时都分别表示一个字符串，如 mytalents[0]和 yourtalents[0]；使用两个下标时都分别表示一个字符，例如 mytalents[1][2]表示 mytalents 数组中第 2 个指针所指向的字符串的第 3 个字符'l'，yourtalents[1][2]表示 youttalentes 数组的第 2 个字符串的第 3 个字符'e'。而且，两者的初始化方式也相同。

但是，它们也有区别。mytalents 数组是一个内含 5 个指针的数组，在我们的系统中共占用 40 字节。而 yourtalents 是一个内含 5 个数组的数组，每个数组内含 40 个 char 类型的值，共占用 200 字节。所以，虽然 mytalents[0]和 yourtalents[0]都分别表示一个字符串，但 mytalents 和 yourtalents 的类型并不相同。mytalents 中的指针指向初始化时所用的字符串字面量的位置，这些字符串字面量被储存在静态内存中；而 yourtalents 中的数组则储存着字符串字面量的副本，所以每个字符串都被储存了两次。此外，为字符串数组分配内存的使用率较低。yourtalents 中的每个元素的大小必须相同，而且必须是能储存最长字符串的大小。

我们可以把 yourtalents 想象成矩形二维数组，每行的长度都是 40 字节；把 mytalents 想象成不规则的数组，每行的长度不同。图 11.2 演示了这两种数组的情况（实际上，mytalents 数组的指针元素所指向的字符串不必储存在连续的内存中，图中所示只是为了强调两种数组的不同）。

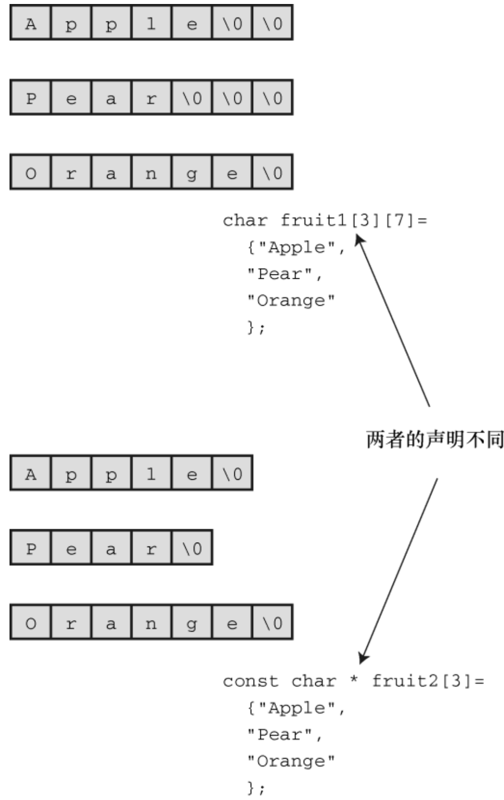


图 11.2　矩形数组和不规则数组

综上所述，如果要用数组表示一系列待显示的字符串，请使用指针数组，因为它比二维字符数组的效率高。但是，指针数组也有自身的缺点。mytalents 中的指针指向的字符串字面量不能更改；而 yourtalentsde 中的内容可以更改。所以，如果要改变字符串或为字符串输入预留空间，不要使用指向字符串字面量的指针。

11.1.2　指针和字符串

读者可能已经注意到了，在讨论字符串时或多或少会涉及指针。实际上，字符串的绝大多数操作都是通过指针完成的。例如，考虑程序清单 11.5 中的程序。

程序清单 11.5　p_and_s.c 程序

```
/* p_and_s.c -- 指针和字符串 */
#include <stdio.h>
int main(void)
{
    const char * mesg = "Don't be a fool!";
    const char * copy;

    copy = mesg;
    printf("%s\n", copy);
    printf("mesg = %s; &mesg = %p; value = %p\n", mesg, &mesg, mesg);
    printf("copy = %s; &copy = %p; value = %p\n", copy, &copy, copy);
```

```
    return 0;
}
```

> **注意**
>
> 如果编译器不识别%p，用%u 或%lu 代替%p。

你可能认为该程序拷贝了字符串"Don't be a fool!"，程序的输出似乎也验证了你的猜测：

```
Don't be a fool!
mesg = Don't be a fool!; &mesg = 0x0012ff48; value = 0x0040a000
copy = Don't be a fool!; &copy = 0x0012ff44; value = 0x0040a000
```

我们来仔细分析最后两个 printf()的输出。首先第 1 项，mesg 和 copy 都以字符串形式输出（%s 转换说明）。这里没问题，两个字符串都是"Don't be a fool!"。

接着第 2 项，打印两个指针的地址。如上输出所示，指针 mesg 和 copy 分别储存在地址为 0x0012ff48 和 0x0012ff44 的内存中。

注意最后一项，显示两个指针的值。所谓指针的值就是它储存的地址。mesg 和 copy 的值都是 0x0040a000，说明它们都指向的同一个位置。因此，程序并未拷贝字符串。语句 copy = mesg;把 mesg 的值赋给 copy，即让 copy 也指向 mesg 指向的字符串。

为什么要这样做？为何不拷贝整个字符串？假设数组有 50 个元素，考虑一下哪种方法更效率：拷贝一个地址还是拷贝整个数组？通常，程序要完成某项操作只需要知道地址就可以了。如果确实需要拷贝整个数组，可以使用 strcpy()或 strncpy()函数，本章稍后介绍这两个函数。

我们已经讨论了如何在程序中定义字符串，接下来看看如何从键盘输入字符串。

11.2 字符串输入

如果想把一个字符串读入程序，首先必须预留储存该字符串的空间，然后用输入函数获取该字符串。

11.2.1 分配空间

要做的第 1 件事是分配空间，以储存稍后读入的字符串。前面提到过，这意味着必须要为字符串分配足够的空间。不要指望计算机在读取字符串时顺便计算它的长度，然后再分配空间（计算机不会这样做，除非你编写一个处理这些任务的函数）。假设编写了如下代码：

```
char *name;
scanf("%s", name);
```

虽然可能会通过编译（编译器很可能给出警告），但是在读入 name 时，name 可能会擦写掉程序中的数据或代码，从而导致程序异常中止。因为 scanf()要把信息拷贝至参数指定的地址上，而此时该参数是个未初始化的指针，name 可能会指向任何地方。大多数程序员都认为出现这种情况很搞笑，但仅限于评价别人的程序时。

最简单的方法是，在声明时显式指明数组的大小：

```
char name[81];
```

现在 name 是一个已分配块（81 字节）的地址。还有一种方法是使用 C 库函数来分配内存，第 12 章将详细介绍。

为字符串分配内存后，便可读入字符串。C 库提供了许多读取字符串的函数：scanf()、gets()和fgets()。我们先讨论最常用 gets()函数。

11.2.2　不幸的 gets()函数

在读取字符串时，scanf()和转换说明%s 只能读取一个单词。可是在程序中经常要读取一整行输入，而不仅仅是一个单词。许多年前，gets()函数就用于处理这种情况。gets()函数简单易用，它读取整行输入，直至遇到换行符，然后丢弃换行符，储存其余字符，并在这些字符的末尾添加一个空字符使其成为一个 C 字符串。它经常和 puts()函数配对使用，该函数用于显示字符串，并在末尾添加换行符。程序清单 11.6 中演示了这两个函数的用法。

程序清单 11.6　getsputs.c 程序

```
/*  getsputs.c  -- 使用 gets() 和 puts() */
#include <stdio.h>
#define STLEN 81
int main(void)
{
    char words[STLEN];

    puts("Enter a string, please.");
    gets(words);  // 典型用法
    printf("Your string twice:\n");
    printf("%s\n", words);
    puts(words);
    puts("Done.");

    return 0;
}
```

下面是该程序在某些编译器（或者至少是旧式编译器）中的运行示例：

```
Enter a string, please.
I want to learn about string theory!
Your string twice:
I want to learn about string theory!
I want to learn about string theory!
Done.
```

整行输入（除了换行符）都被储存在 words 中，puts(words)和 printf("%s\n, words")的效果相同。

下面是该程序在另一个编译器中的输出示例：

```
Enter a string, please.
warning: this program uses gets(), which is unsafe.
Oh, no!
Your string twice:
Oh, no!
Oh, no!
Done.
```

编译器在输出中插入了一行警告消息。每次运行这个程序，都会显示这行消息。但是，并非所有的编译器都会这样做。其他编译器可能在编译过程中给出警告，但不会引起你的注意。

这是怎么回事？问题出在 gets()唯一的参数是 words，它无法检查数组是否装得下输入行。上一章

介绍过，数组名会被转换成该数组首元素的地址，因此，gets()函数只知道数组的开始处，并不知道数组中有多少个元素。

如果输入的字符串过长，会导致缓冲区溢出（*buffer overflow*），即多余的字符超出了指定的目标空间。如果这些多余的字符只是占用了尚未使用的内存，就不会立即出现问题；如果它们擦写掉程序中的其他数据，会导致程序异常中止；或者还有其他情况。为了让输入的字符串容易溢出，把程序中的 STLEN 设置为 5，程序的输出如下：

```
Enter a string, please.
warning: this program uses gets(), which is unsafe.
I think I'll be just fine.
Your string twice:
I think I'll be just fine.
I think I'll be just fine.
Done.
Segmentation fault: 11
```

“Segmentation fault”（分段错误）似乎不是个好提示，的确如此。在 UNIX 系统中，这条消息说明该程序试图访问未分配的内存。

C 提供解决某些编程问题的方法可能会导致陷入另一个尴尬棘手的困境。但是，为什么要特别提到 gets()函数？因为该函数的不安全行为造成了安全隐患。过去，有些人通过系统编程，利用 gets()插入和运行一些破坏系统安全的代码。

不久，C 编程社区的许多人都建议在编程时摒弃 gets()。制定 C99 标准的委员会把这些建议放入了标准，承认了 gets()的问题并建议不要再使用它。尽管如此，在标准中保留 gets()也合情合理，因为现有程序中含有大量使用该函数的代码。而且，只要使用得当，它的确是一个很方便的函数。

好景不长，C11 标准委员会采取了更强硬的态度，直接从标准中废除了 gets()函数。既然标准已经发布，那么编译器就必须根据标准来调整支持什么，不支持什么。然而在实际应用中，编译器为了能兼容以前的代码，大部分都继续支持 gets()函数。不过，我们使用的编译器，可没那么大方。

11.2.3 gets()的替代品

过去通常用 fgets()来代替 gets()，fgets()函数稍微复杂些，在处理输入方面与 gets()略有不同。C11 标准新增的 gets_s()函数也可代替 gets()。该函数与 gets()函数更接近，而且可以替换现有代码中的 gets()。但是，它是 stdio.h 输入/输出函数系列中的可选扩展，所以支持 C11 的编译器也不一定支持它。

1. fgets()函数（和 fputs()）

fgets()函数通过第 2 个参数限制读入的字符数来解决溢出的问题。该函数专门设计用于处理文件输入，所以一般情况下可能不太好用。fgets()和 gets()的区别如下。

- fgets()函数的第 2 个参数指明了读入字符的最大数量。如果该参数的值是 n，那么 fgets()将读入 n-1 个字符，或者读到遇到的第一个换行符为止。
- 如果 fgets()读到一个换行符，会把它储存在字符串中。这点与 gets()不同，gets()会丢弃换行符。
- fgets()函数的第 3 个参数指明要读入的文件。如果读入从键盘输入的数据，则以 stdin（标准输入）作为参数，该标识符定义在 stdio.h 中。

因为 fgets()函数把换行符放在字符串的末尾（假设输入行不溢出），通常要与 fputs()函数（和

puts()类似）配对使用，除非该函数不在字符串末尾添加换行符。fputs()函数的第 2 个参数指明它要写入的文件。如果要显示在计算机显示器上，应使用 stdout（标准输出）作为该参数。程序清单 11.7 演示了 fgets()和 fputs()函数的用法。

程序清单 11.7　fgets1.c 程序

```
/*  fgets1.c  -- 使用 fgets() 和 fputs() */
#include <stdio.h>
#define STLEN 14
int main(void)
{
    char words[STLEN];

    puts("Enter a string, please.");
    fgets(words, STLEN, stdin);
    printf("Your string twice (puts(), then fputs()):\n");
    puts(words);
    fputs(words, stdout);
    puts("Enter another string, please.");
    fgets(words, STLEN, stdin);
    printf("Your string twice (puts(), then fputs()):\n");
    puts(words);
    fputs(words, stdout);
    puts("Done.");

    return 0;
}
```

下面是该程序的输出示例：

```
Enter a string, please.
apple pie
Your string twice (puts(), then fputs()):
apple pie

apple pie
Enter another string, please.
strawberry shortcake
Your string twice (puts(), then fputs()):
strawberry sh
strawberry shDone.
```

第 1 行输入，apple pie，比 fgets()读入的整行输入短，因此，apple pie\n\0 被储存在数组中。所以当 puts()显示该字符串时又在末尾添加了换行符，因此 apple pie 后面有一行空行。因为 fputs()不在字符串末尾添加换行符，所以并未打印出空行。

第 2 行输入，strawberry shortcake，超过了大小的限制，所以 fgets()只读入了 13 个字符，并把 strawberry sh\0 储存在数组中。再次提醒读者注意，puts()函数会在待输出字符串末尾添加一个换行符，而 fputs()不会这样做。

fputs()函数返回指向 char 的指针。如果一切进行顺利，该函数返回的地址与传入的第 1 个参数相同。但是，如果函数读到文件结尾，它将返回一个特殊的指针：空指针（*null pointer*）。该指针保证不会指向有效的数据，所以可用于标识这种特殊情况。在代码中，可以用数字 0 来代替，不过在 C 语言中用宏 NULL 来代替更常见（如果在读入数据时出现某些错误，该函数也返回 NULL）。程序清单 11.8 演示

了一个简单的循环，读入并显示用户输入的内容，直到 fgets()读到文件结尾或空行（即，首字符是换行符）。

程序清单 11.8 fgets2.c **程序**

```
/*  fgets2.c  -- 使用 fgets() 和 fputs() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
    char words[STLEN];

    puts("Enter strings (empty line to quit):");
    while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
        fputs(words, stdout);
    puts("Done.");

    return 0;
}
```

下面是该程序的输出示例：

```
Enter strings (empty line to quit):
By the way, the gets() function
By the way, the gets() function
also returns a null pointer if it
also returns a null pointer if it
encounters end-of-file.
encounters end-of-file.

Done.
```

有意思，虽然 STLEN 被设置为 10，但是该程序似乎在处理过长的输入时完全没问题。程序中的 fgets()一次读入 STLEN - 1 个字符（该例中为 9 个字符）。所以，一开始它只读入了“By the wa”，并储存为 By the wa\0；接着 fputs()打印该字符串，而且并未换行。然后 while 循环进入下一轮迭代，fgets()继续从剩余的输入中读入数据，即读入“y, the ge”并储存为 y, the ge\0；接着 fputs()在刚才打印字符串的这一行接着打印第 2 次读入的字符串。然后 while 进入下一轮迭代，fgets()继续读取输入、fputs()打印字符串，这一过程循环进行，直到读入最后的“tion\n”。fgets()将其储存为 tion\n\0，fputs()打印该字符串，由于字符串中的\n，光标被移至下一行开始处。

系统使用缓冲的 I/O。这意味着用户在按下 Return 键之前，输入都被储存在临时存储区（即，缓冲区）中。按下 Return 键就在输入中增加了一个换行符，并把整行输入发送给 fgets()。对于输出，fputs()把字符发送给另一个缓冲区，当发送换行符时，缓冲区中的内容被发送至屏幕上。

fgets()储存换行符有好处也有坏处。坏处是你可能并不想把换行符储存在字符串中，这样的换行符会带来一些麻烦。好处是对于储存的字符串而言，检查末尾是否有换行符可以判断是否读取了一整行。如果不是一整行，要妥善处理一行中剩下的字符。

首先，如何处理掉换行符？一个方法是在已储存的字符串中查找换行符，并将其替换成空字符：

```
while (words[i] != '\n') // 假设\n 在 words 中
    i++;
words[i] = '\0';
```

其次，如果仍有字符串留在输入行怎么办？一个可行的办法是，如果目标数组装不下一整行输入，就

丢弃那些多出的字符：

```
while (getchar() != '\n')  // 读取但不储存输入，包括\n
    continue;
```

程序清单 11.9 在程序清单 11.8 的基础上添加了一部分测试代码。该程序读取输入行，删除储存在字符串中的换行符，如果没有换行符，则丢弃数组装不下的字符。

程序清单 11.9　fgets3.c 程序

```
/*  fgets3.c  -- 使用 fgets() */
#include <stdio.h>
#define STLEN 10
int main(void)
{
    char words[STLEN];
    int i;

    puts("Enter strings (empty line to quit):");
    while (fgets(words, STLEN, stdin) != NULL && words[0] != '\n')
    {
        i = 0;
        while (words[i] != '\n' && words[i] != '\0')
            i++;
        if (words[i] == '\n')
            words[i] = '\0';
        else  // 如果 word[i] == '\0'则执行这部分代码
            while (getchar() != '\n')
                 continue;
        puts(words);
    }
    puts("done");
    return 0;
}
```

循环

```
while (words[i] != '\n' && words[i] != '\0')
    i++;
```

遍历字符串，直至遇到换行符或空字符。如果先遇到换行符，下面的 if 语句就将其替换成空字符；如果先遇到空字符，else 部分便丢弃输入行的剩余字符。下面是该程序的输出示例：

```
Enter strings (empty line to quit):
This
This
program seems
program s
unwilling to accept long lines.
unwilling
But it doesn't get stuck on long
But it do
lines either.
lines eit

done
```

空字符和空指针

程序清单 11.9 中出现了空字符和空指针。从概念上看，两者完全不同。空字符（或'\0'）是用于标记 C 字符串末尾的字符，其对应字符编码是 0。由于其他字符的编码不可能是 0，所以不可能是字符串的一部分。

空指针（或 NULL）有一个值，该值不会与任何数据的有效地址对应。通常，函数使用它返回一个有效地址表示某些特殊情况发生，例如遇到文件结尾或未能按预期执行。

空字符是整数类型，而空指针是指针类型。两者有时容易混淆的原因是：它们都可以用数值 0 来表示。但是，从概念上看，两者是不同类型的 0。另外，空字符是一个字符，占 1 字节；而空指针是一个地址，通常占 4 字节。

2. gets_s()函数

C11 新增的 gets_s()函数（可选）和 fgets()类似，用一个参数限制读入的字符数。假设把程序清单 11.9 中的 fgets()换成 gets_s()，其他内容不变，那么下面的代码将把一行输入中的前 9 个字符读入 words 数组中，假设末尾有换行符：

```
gets_s(words, STLEN);
```

gets_s()与 fgets()的区别如下。

- gets_s()只从标准输入中读取数据，所以不需要第 3 个参数。
- 如果 gets_s()读到换行符，会丢弃它而不是储存它。
- 如果 gets_s()读到最大字符数都没有读到换行符，会执行以下几步。首先把目标数组中的首字符设置为空字符，读取并丢弃随后的输入直至读到换行符或文件结尾，然后返回空指针。接着，调用依赖实现的“处理函数”（或你选择的其他函数），可能会中止或退出程序。

第 2 个特性说明，只要输入行未超过最大字符数，gets_s()和 gets()几乎一样，完全可以用 gets_s()替换 gets()。第 3 个特性说明，要使用这个函数还需要进一步学习。

我们来比较一下 gets()、fgets()和 gets_s()的适用性。如果目标存储区装得下输入行，3 个函数都没问题。但是 fgets()会保留输入末尾的换行符作为字符串的一部分，要编写额外的代码将其替换成空字符。

如果输入行太长会怎样？使用 gets()不安全，它会擦写现有数据，存在安全隐患。gets_s()函数很安全，但是，如果并不希望程序中止或退出，就要知道如何编写特殊的“处理函数”。另外，如果打算让程序继续运行，gets_s()会丢弃该输入行的其余字符，无论你是否需要。由此可见，当输入太长，超过数组可容纳的字符数时，fgets()函数最容易使用，而且可以选择不同的处理方式。如果要让程序继续使用输入行中超出的字符，可以参考程序清单 11.8 中的处理方法。如果想丢弃输入行的超出字符，可以参考程序清单 11.9 中的处理方法。

所以，当输入与预期不符时，gets_s()完全没有 fgets()函数方便、灵活。也许这也是 gets_s()只作为 C 库的可选扩展的原因之一。鉴于此，fgets()通常是处理类似情况的最佳选择。

3. s_gets()函数

程序清单 11.9 演示了 fgets()函数的一种用法：读取整行输入并用空字符代替换行符，或者读取一部分输入，并丢弃其余部分。既然没有处理这种情况的标准函数，我们就创建一个，在后面的程序中会用得上。程序清单 11.10 提供了一个这样的函数。

程序清单 11.10　s_gets()函数

```
char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)  // 即，ret_val != NULL
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

如果 fgets()返回 NULL，说明读到文件结尾或出现读取错误，s_gets()函数跳过了这个过程。它模仿程序清单 11.9 的处理方法，如果字符串中出现换行符，就用空字符替换它；如果字符串中出现空字符，就丢弃该输入行的其余字符，然后返回与 fgets()相同的值。我们在后面的示例中将讨论 fgets()函数。

也许读者想了解为什么要丢弃过长输入行中的余下字符。这是因为，输入行中多出来的字符会被留在缓冲区中，成为下一次读取语句的输入。例如，如果下一条读取语句要读取的是 double 类型的值，就可能导致程序崩溃。丢弃输入行余下的字符保证了读取语句与键盘输入同步。

我们设计的 s_gets()函数并不完美，它最严重的缺陷是遇到不合适的输入时毫无反应。它丢弃多余的字符时，既不通知程序也不告知用户。但是，用来替换前面程序示例中的 gets()足够了。

11.2.4　scanf()函数

我们再来研究一下scanf()。前面的程序中用scanf()和%s转换说明读取字符串。scanf()和gets()或 fgets()的区别在于它们如何确定字符串的末尾：scanf()更像是“获取单词”函数，而不是“获取字符串”函数；如果预留的存储区装得下输入行，gets()和 fgets()会读取第 1 个换行符之前所有的字符。scanf()函数有两种方法确定输入结束。无论哪种方法，都从第 1 个非空白字符作为字符串的开始。如果使用%s 转换说明，以下一个空白字符（空行、空格、制表符或换行符）作为字符串的结束（字符串不包括空白字符）。如果指定了字段宽度，如%10s，那么 scanf()将读取 10 个字符或读到第 1 个空白字符停止（先满足的条件即是结束输入的条件），见图 11.3。

输入语句	原输入序列*	name中的内容	剩余输入序列
scanf('%s", name);	Fleebert□Hup	Fleebert	□Hup
scanf('%5s", name);	Fleebert□Hup	Fleeb	ert□Hup
scanf('%5s", name);	Ann□Ular	Ann	□Ular

*□表示空格字符

图 11.3　字段宽度和 scanf()

前面介绍过，scanf()函数返回一个整数值，该值等于 scanf()成功读取的项数或 EOF（读到文件结尾时返回 EOF）。

程序清单 11.11 演示了在 scanf()函数中指定字段宽度的用法。

程序清单 11.11　scan_str.c 程序

```
/* scan_str.c -- 使用 scanf() */
#include <stdio.h>
int main(void)
{
    char name1[11], name2[11];
    int count;

    printf("Please enter 2 names.\n");
    count = scanf("%5s %10s", name1, name2);
    printf("I read the %d names %s and %s.\n", count, name1, name2);

    return 0;
}
```

下面是该程序的 3 个输出示例：

```
Please enter 2 names.
Jesse Jukes
I read the 2 names Jesse and Jukes.

Please enter 2 names.
Liza Applebottham
I read the 2 names Liza and Applebotth.

Please enter 2 names.
Portensia Callowit
I read the 2 names Porte and nsia.
```

第 1 个输出示例，两个名字的字符个数都未超过字段宽度。第 2 个输出示例，只读入了 Applebottham 的前 10 个字符 Applebotth（因为使用了%10s 转换说明）。第 3 个输出示例，Portensia 的后 4 个字符 nsia 被写入 name2 中，因为第 2 次调用 scanf()时，从上一次调用结束的地方继续读取数据。在该例中，读取的仍是 Portensia 中的字母。

根据输入数据的性质，用 fgets()读取从键盘输入的数据更合适。例如，scanf()无法完整读取书名或歌曲名，除非这些名称是一个单词。scanf()的典型用法是读取并转换混合数据类型为某种标准形式。例如，如果输入行包含一种工具名、库存量和单价，就可以使用 scanf()。否则可能要自己拼凑一个函数处理一些输入检查。如果一次只输入一个单词，用 scanf()也没问题。

scanf()和 gets()类似，也存在一些潜在的缺点。如果输入行的内容过长，scanf()也会导致数据溢出。不过，在%s 转换说明中使用字段宽度可防止溢出。

11.3　字符串输出

讨论完字符串输入，接下来我们讨论字符串输出。C 有 3 个标准库函数用于打印字符串：put()、fputs()和 printf()。

11.3.1　puts()函数

puts()函数很容易使用，只需把字符串的地址作为参数传递给它即可。程序清单 11.12 演示了 puts()的一些用法。

程序清单 11.12　put_out.c **程序**

```
/* put_out.c -- 使用 puts() */
#include <stdio.h>
#define DEF "I am a #defined string."
int main(void)
{
    char str1[80] = "An array was initialized to me.";
    const char * str2 = "A pointer was initialized to me.";

    puts("I'm an argument to puts().");
    puts(DEF);
    puts(str1);
    puts(str2);
    puts(&str1[5]);
    puts(str2 + 4);

    return 0;
}
```

该程序的输出如下：

```
I'm an argument to puts().
I am a #defined string.
An array was initialized to me.
A pointer was initialized to me.
ray was initialized to me.
inter was initialized to me.
```

如上所示，每个字符串独占一行，因为 puts()在显示字符串时会自动在其末尾添加一个换行符。

该程序示例再次说明，用双引号括起来的内容是字符串常量，且被视为该字符串的地址。另外，储存字符串的数组名也被看作是地址。在第 5 个 puts()调用中，表达式&str1[5]是 str1 数组的第 6 个元素（r），puts()从该元素开始输出。与此类似，第 6 个 puts()调用中，str2+4 指向储存"pointer"中 i 的存储单元，puts()从这里开始输出。

puts()如何知道在何处停止？该函数在遇到空字符时就停止输出，所以必须确保有空字符。不要模仿程序清单 11.13 中的程序！

程序清单 11.13　nono.c **程序**

```
/* nono.c -- 千万不要模仿！ */
#include <stdio.h>
int main(void)
{
    char side_a[] = "Side A";
    char dont[] = { 'W', 'O', 'W', '!' };
    char side_b[] = "Side B";

    puts(dont); /* dont 不是一个字符串 */

    return 0;
}
```

由于 dont 缺少一个表示结束的空字符，所以它不是一个字符串，因此 puts()不知道在何处停止。它会一直打印 dont 后面内存中的内容，直到发现一个空字符为止。为了让 puts()能尽快读到空字符，我们把 dont 放在 side_a 和 side_b 之间。下面是该程序的一个运行示例：

```
WOW!Side A
```

我们使用的编译器把 side_a 数组储存在 dont 数组之后，所以 puts()一直输出至遇到 side_a 中的空字符。你所使用的编译器输出的内容可能不同，这取决于编译器如何在内存中储存数据。如果删除程序中的 side_a 和 side_b 数组会怎样？通常内存中有许多空字符，如果幸运的话，puts()很快就会发现一个。但是，这样做很不靠谱。

11.3.2 fputs()函数

fputs()函数是 puts()针对文件定制的版本。它们的区别如下。

- fputs()函数的第 2 个参数指明要写入数据的文件。如果要打印在显示器上，可以用定义在 stdio.h 中的 stdout（标准输出）作为该参数。
- 与 puts()不同，fputs()不会在输出的末尾添加换行符。

注意，gets()丢弃输入中的换行符，但是 puts()在输出中添加换行符。另一方面，fgets()保留输入中的换行符，fputs()不在输出中添加换行符。假设要编写一个循环，读取一行输入，另起一行打印出该输入。可以这样写：

```
char line[81];
while (gets(line))// 与 while (gets(line) != NULL)相同
    puts(line);
```

如果 gets()读到文件结尾会返回空指针。对空指针求值为 0（即为假），这样便可结束循环。或者，可以这样写：

```
char line[81];
while (fgets(line, 81, stdin))
    fputs(line, stdout);
```

第 1 个循环（使用 gets()和 puts()的 while 循环），line 数组中的字符串显示在下一行，因为 puts()在字符串末尾添加了一个换行符。第 2 个循环（使用 fgets()和 fputs()的 while 循环），line 数组中的字符串也显示在下一行，因为 fgets()把换行符储存在字符串末尾。注意，如果混合使用 fgets()输入和 puts()输出，每个待显示的字符串末尾就会有两个换行符。这里关键要注意：puts()应与 gets()配对使用，fputs()应与 fgets()配对使用。

我们在这里提到已被废弃的 gets()，并不是鼓励使用它，而是为了让读者了解它的用法。如果今后遇到包含该函数的代码，不至于看不懂。

11.3.3 printf()函数

在第 4 章中，我们详细讨论过 printf()函数的用法。和 puts()一样，printf()也把字符串的地址作为参数。printf()函数用起来没有 puts()函数那么方便，但是它更加多才多艺，因为它可以格式化不同的数据类型。

与 puts()不同的是，printf()不会自动在每个字符串末尾加上一个换行符。因此，必须在参数中指明应该在哪里使用换行符。例如：

```
printf("%s\n", string);
```

和下面的语句效果相同：

```
puts(string);
```

如上所示，printf()的形式更复杂些，需要输入更多代码，而且计算机执行的时间也更长（但是你觉察不到）。然而，使用 printf()打印多个字符串更加简单。例如，下面的语句把 Well、用户名和一个#define 定义的字符串打印在一行：

```
printf("Well, %s, %s\n", name, MSG);
```

11.4　自定义输入/输出函数

不一定非要使用 C 库中的标准函数，如果无法使用这些函数或者不想用它们，完全可以在 getchar()和putchar()的基础上自定义所需的函数。假设你需要一个类似puts()但是不会自动添加换行符的函数。程序清单 11.14 给出了一个这样的函数。

程序清单 11.14　put1()函数

```
/* put1.c -- 打印字符串，不添加\n */
#include <stdio.h>
void put1(const char * string)/* 不会改变字符串 */
{
    while (*string != '\0')
        putchar(*string++);
}
```

指向 char 的指针 string 最初指向传入参数的首元素。因为该函数不会改变传入的字符串，所以形参使用了 const 限定符。打印了首元素的内容后，指针递增 1，指向下一个元素。while 循环重复这一过程，直到指针指向包含空字符的元素。记住，++的优先级高于*，因此 putchar(*string++)打印 string 指向的值，递增的是 string 本身，而不是递增它所指向的字符。

可以把 put1.c 程序作为编写字符串处理函数的模型。因为每个字符串都以空字符结尾，所以不用给函数传递字符串的大小。函数依次处理每个字符，直至遇到空字符。

用数组表示法编写这个函数稍微复杂些：

```
int i = 0;
while (string[i]!= '\0')
        putchar(string[i++]);
```

要为数组索引创建一个额外的变量。

许多 C 程序员会在 while 循环中使用下面的测试条件：

```
while (*string)
```

当 string 指向空字符时，*string 的值是 0，即测试条件为假，while 循环结束。这种方法比上面两种方法简洁。但是，如果不熟悉 C 语言，可能觉察不出来。这种处理方法很普遍，作为 C 程序员应该熟悉这种写法。

注意

为什么程序清单 11.14 中的形式参数是 const char * string，而不是 const char sting[]？从技术方面看，两者等价且都有效。使用带方括号的写法是为了提醒用户：该函数处理的是数组。然而，如果要处理字符串，实际参数可以是数组名、用双引号括起来的字符串，或声明为 char *类型的变量。用 const char * string 可以提醒用户：实际参数不一定是数组。

假设要设计一个类似 puts() 的函数，而且该函数还给出待打印字符的个数。如程序清单 11.15 所示，添加一个功能很简单。

程序清单 11.15　put2.c 程序

```
/* put2.c -- 打印一个字符串，并统计打印的字符数 */
#include <stdio.h>
int put2(const char * string)
{
    int count = 0;
    while (*string)    /* 常规用法 */
    {
        putchar(*string++);
        count++;
    }
    putchar('\n');     /* 不统计换行符 */

    return(count);
}
```

下面的函数调用将打印字符串 pizza：

```
put1("pizza");
```

下面的调用将返回统计的字符数，并将其赋给 num（该例中，num 的值是 5）：

```
num = put2("pizza");
```

程序清单 11.16 使用一个简单的驱动程序测试 put1() 和 put2()，并演示了嵌套函数的调用。

程序清单 11.16　.c 程序

```
//put_put.c -- 用户自定义输出函数
#include <stdio.h>
void put1(const char *);
int put2(const char *);

int main(void)
{
    put1("If I'd as much money");
    put1(" as I could spend,\n");
    printf("I count %d characters.\n",
        put2("I never would cry old chairs to mend."));

    return 0;
}

void put1(const char * string)
{
    while (*string)  /* 与 *string != '\0' 相同 */
        putchar(*string++);
}

int put2(const char * string)
{
    int count = 0;
    while (*string)
    {
```

```
        putchar(*string++);
        count++;
    }
    putchar('\n');

    return(count);
}
```

程序中使用 printf()打印 put2()的值，但是为了获得 put2()的返回值，计算机必须先执行 put2()，因此在打印字符数之前先打印了传递给该函数的字符串。下面是该程序的输出：

```
If I'd as much money as I could spend,
I never would cry old chairs to mend.
I count 37 characters.
```

11.5　字符串函数

C 库提供了多个处理字符串的函数，ANSI C 把这些函数的原型放在 string.h 头文件中。其中最常用的函数有 strlen()、strcat()、strcmp()、strncmp()、strcpy()和 strncpy()。另外，还有 sprintf()函数，其原型在 stdio.h 头文件中。欲了解 string.h 系列函数的完整列表，请查阅附录 B 中的参考资料 V“新增 C99 和 C11 的标准 ANSI C 库”。

11.5.1　strlen()函数

strlen()函数用于统计字符串的长度。下面的函数可以缩短字符串的长度，其中用到了 strlen()：

```
void fit(char *string, unsigned int size)
{
    if (strlen(string) > size)
        string[size] = '\0';
}
```

该函数要改变字符串，所以函数头在声明形式参数 string 时没有使用 const 限定符。

程序清单 11.17 中的程序测试了 fit()函数。注意代码中使用了 C 字符串常量的串联特性。

程序清单 11.17　test_fit.c 程序

```
/* test_fit.c -- 使用缩短字符串长度的函数 */
#include <stdio.h>
#include <string.h>    /* 内含字符串函数原型 */
void fit(char *, unsigned int);

int main(void)
{
    char mesg [] = "Things should be as simple as possible,"
                            " but not simpler.";

    puts(mesg);
    fit(mesg, 38);
    puts(mesg);
    puts("Let's look at some more of the string.");
    puts(mesg + 39);

    return 0;
}
```

```
void fit(char *string, unsigned int size)
{
    if (strlen(string) > size)
        string[size] = '\0';
}
```

下面是该程序的输出：

```
Things should be as simple as possible, but not simpler.
Things should be as simple as possible
Let's look at some more of the string.
but not simpler.
```

fit()函数把第 39 个元素的逗号替换成'\0'字符。puts()函数在空字符处停止输出，并忽略其余字符。然而，这些字符还在缓冲区中，下面的函数调用把这些字符打印了出来：

```
puts(mesg + 8);
```

表达式 mesg + 39 是 mesg[39]的地址，该地址上储存的是空格字符。所以 put()显示该字符并继续输出直至遇到原来字符串中的空字符。图 11.4 演示了这一过程。

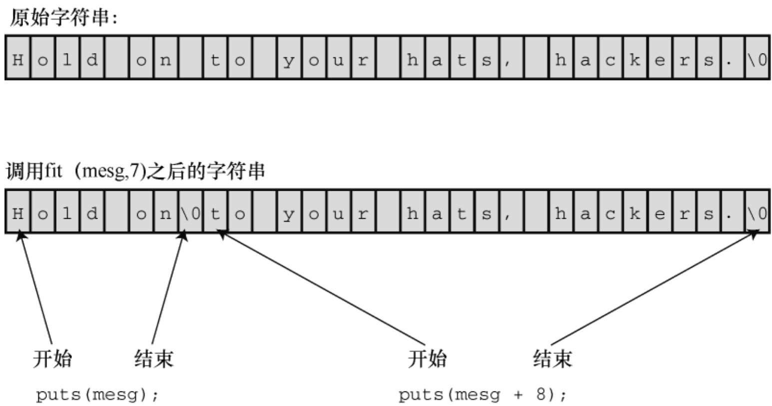


图 11.4 puts()函数和空字符

> **注意**
>
> 一些 ANSI 之前的系统使用 strings.h 头文件，而有些系统可能根本没有字符串头文件。

string.h 头文件中包含了 C 字符串函数系列的原型，因此程序清单 11.17 要包含该头文件。

11.5.2 strcat()函数

strcat()（用于拼接字符串）函数接受两个字符串作为参数。该函数把第 2 个字符串的备份附加在第 1 个字符串末尾，并把拼接后形成的新字符串作为第 1 个字符串，第 2 个字符串不变。strcat()函数的类型是 char *（即，指向 char 的指针）。strcat()函数返回第 1 个参数，即拼接第 2 个字符串后的第 1 个字符串的地址。

程序清单 11.18 演示了 strcat()的用法。该程序还使用了程序清单 11.10 的 s_gets()函数。回忆一下，该函数使用 fgets()读取一整行，如果有换行符，将其替换成空字符。

程序清单 11.18　str_cat.c 程序

```
/* str_cat.c -- 拼接两个字符串 */
#include <stdio.h>
#include <string.h>  /* strcat()函数的原型在该头文件中 */
#define SIZE 80
char * s_gets(char * st, int n);
int main(void)
{
    char flower[SIZE];
    char addon [] = "s smell like old shoes.";

    puts("What is your favorite flower?");
    if (s_gets(flower, SIZE))
    {
        strcat(flower, addon);
        puts(flower);
        puts(addon);
    }
    else
        puts("End of file encountered!");
    puts("bye");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                 continue;
    }
    return ret_val;
}
```

该程序的输出示例如下：

```
What is your favorite flower?
wonderflower
wonderflowers smell like old shoes.
s smell like old shoes.
bye
```

从以上输出可以看出，flower 改变了，而 addon 保持不变。

11.5.3 strncat()函数

strcat()函数无法检查第 1 个数组是否能容纳第 2 个字符串。如果分配给第 1 个数组的空间不够大，多出来的字符溢出到相邻存储单元时就会出问题。当然，可以像程序清单 11.15 那样，用 strlen()查看第 1 个数组的长度。注意，要给拼接后的字符串长度加 1 才够空间存放末尾的空字符。或者，用 strncat()，该函数的第 3 个参数指定了最大添加字符数。例如，strncat(bugs, addon, 13)将把 addon 字符串的内容附加给 bugs，在加到第 13 个字符或遇到空字符时停止。因此，算上空字符（无论哪种情况都要添加空字符），bugs 数组应该足够大，以容纳原始字符串（不包含空字符）、添加原始字符串在后面的 13 个字符和末尾的空字符。程序清单 11.19 使用这种方法，计算 avaiable 变量的值，用于表示允许添加的最大字符数。

程序清单 11.19 join_chk.c 程序

```c
/* join_chk.c -- 拼接两个字符串，检查第 1 个数组的大小 */
#include <stdio.h>
#include <string.h>
#define SIZE 30
#define BUGSIZE 13
char * s_gets(char * st, int n);
int main(void)
{
    char flower[SIZE];
    char addon [] = "s smell like old shoes.";
    char bug[BUGSIZE];
    int available;

    puts("What is your favorite flower?");
    s_gets(flower, SIZE);
    if ((strlen(addon) + strlen(flower) + 1) <= SIZE)
        strcat(flower, addon);
    puts(flower);
    puts("What is your favorite bug?");
    s_gets(bug, BUGSIZE);
    available = BUGSIZE - strlen(bug) - 1;
    strncat(bug, addon, available);
    puts(bug);

    return 0;
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
```

```
                continue;
    }
    return ret_val;
}
```

下面是该程序的运行示例：

```
What is your favorite flower?
Rose
Roses smell like old shoes.
What is your favorite bug?
Aphid
Aphids smell
```

读者可能已经注意到，strcat()和 gets()类似，也会导致缓冲区溢出。为什么 C11 标准不废弃 strcat()，只留下 strncat()？为何对 gets()那么残忍？这也许是因为 gets()造成的安全隐患来自于使用该程序的人，而 strcat()暴露的问题是那些粗心的程序员造成的。无法控制用户会进行什么操作，但是，可以控制你的程序做什么。C 语言相信程序员，因此程序员有责任确保 strcat()的使用安全。

11.5.4　strcmp()函数

假设要把用户的响应与已储存的字符串作比较，如程序清单 11.20 所示。

程序清单 11.20　nogo.c **程序**

```
/* nogo.c -- 该程序是否能正常运行？ */
#include <stdio.h>
#define ANSWER "Grant"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
    char try[SIZE];

    puts("Who is buried in Grant's tomb?");
    s_gets(try, SIZE);
    while (try != ANSWER)
    {
        puts("No, that's wrong. Try again.");
        s_gets(try, SIZE);
    }
    puts("That's right!");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
```

```
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

这个程序看上去没问题，但是运行后却不对劲。ANSWER 和 try 都是指针，所以 try != ANSWER 检查的不是两个字符串是否相等，而是这两个字符串的地址是否相同。因为 ANSWE 和 try 储存在不同的位置，所以这两个地址不可能相同，因此，无论用户输入什么，程序都提示输入不正确。这真让人沮丧。

该函数要比较的是字符串的内容，不是字符串的地址。读者可以自己设计一个函数，也可以使用 C 标准库中的 strcmp()函数（用于字符串比较）。该函数通过比较运算符来比较字符串，就像比较数字一样。如果两个字符串参数相同，该函数就返回 0，否则返回非零值。修改后的版本如程序清单 11.21 所示。

程序清单 11.21　compare.c 程序

```
/* compare.c -- 该程序可以正常运行 */
#include <stdio.h>
#include <string.h>   // strcmp()函数的原型在该头文件中

#define ANSWER "Grant"
#define SIZE 40
char * s_gets(char * st, int n);

int main(void)
{
    char try[SIZE];

    puts("Who is buried in Grant's tomb?");
    s_gets(try, SIZE);
    while (strcmp(try, ANSWER) != 0)
    {
        puts("No, that's wrong. Try again.");
        s_gets(try, SIZE);
    }
    puts("That's right!");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
```

```
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

注意

由于非零值都为“真”，所以许多经验丰富的C程序员会把该例main()中的while循环头写成：

while (strcmp(try, ANSWER))

strcmp()函数比较的是字符串，不是整个数组，这是非常好的功能。虽然数组try占用了40字节，而储存在其中的"Grant"只占用了6字节（还有一个用来放空字符），strcmp()函数只会比较try中第1个空字符前面的部分。所以，可以用strcmp()比较储存在不同大小数组中的字符串。

如果用户输入GRANT、grant或Ulysses S. Grant会怎样？程序会告知用户输入错误。希望程序更友好，必须把所有正确答案的可能性包含其中。这里可以使用一些小技巧。例如，可以使用#define定义类似GRANT这样的答案，并编写一个函数把输入的内容都转换成小写，就解决了大小写的问题。但是，还要考虑一些其他错误的形式，这些留给读者完成。

1. strcmp()的返回值

如果strcmp()比较的字符串不同，它会返回什么值？请看程序清单11.22的程序示例。

程序清单 11.22　compback.c 程序

```
/* compback.c -- strcmp()的返回值 */
#include <stdio.h>
#include <string.h>
int main(void)
{

    printf("strcmp(\"A\", \"A\") is ");
    printf("%d\n", strcmp("A", "A"));

    printf("strcmp(\"A\", \"B\") is ");
    printf("%d\n", strcmp("A", "B"));

    printf("strcmp(\"B\", \"A\") is ");
    printf("%d\n", strcmp("B", "A"));

    printf("strcmp(\"C\", \"A\") is ");
    printf("%d\n", strcmp("C", "A"));

    printf("strcmp(\"Z\", \"a\") is ");
    printf("%d\n", strcmp("Z", "a"));

    printf("strcmp(\"apples\", \"apple\") is ");
    printf("%d\n", strcmp("apples", "apple"));

    return 0;
}
```

在我们的系统中运行该程序，输出如下：

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 1
strcmp("Z", "a") is -1
strcmp("apples", "apple") is 1
```

strcmp()比较"A"和本身，返回 0；比较"A"和"B"，返回-1；比较"B"和"A"，返回 1。这说明，如果在字母表中第 1 个字符串位于第 2 个字符串前面，strcmp()中就返回负数；反之，strcmp()则返回正数。所以，strcmp()比较"C"和"A"，返回 1。其他系统可能返回 2，即两者的 ASCII 码之差。ASCII 标准规定，在字母表中，如果第 1 个字符串在第 2 个字符串前面，strcmp()返回一个负数；如果两个字符串相同，strcmp()返回 0；如果第 1 个字符串在第 2 个字符串后面，strcmp()返回正数。然而，返回的具体值取决于实现。例如，下面给出在不同实现中的输出，该实现返回两个字符的差值：

```
strcmp("A", "A") is 0
strcmp("A", "B") is -1
strcmp("B", "A") is 1
strcmp("C", "A") is 2
strcmp("Z", "a") is -7
strcmp("apples", "apple") is 115
```

如果两个字符串开始的几个字符都相同会怎样？一般而言，strcmp()会依次比较每个字符，直到发现第 1 对不同的字符为止。然后，返回相应的值。例如，在上面的最后一个例子中，"apples"和"apple"只有最后一对字符不同（"apples"的 s 和"apple"的空字符）。由于空字符在 ASCII 中排第 1。字符 s 一定在它后面，所以 strcmp()返回一个正数。

最后一个例子表明，strcmp()比较所有的字符，不只是字母。所以，与其说该函数按字母顺序进行比较，不如说是按机器排序序列（*machine collating sequence*）进行比较，即根据字符的数值进行比较（通常都使用 ASCII 值）。在 ASCII 中，大写字母在小写字母前面，所以 strcmp("Z", "a")返回的是负值。

大多数情况下，strcmp()返回的具体值并不重要，我们只在意该值是 0 还是非 0（即，比较的两个字符串是否相等）。或者按字母排序字符串，在这种情况下，需要知道比较的结果是为正、为负还是为 0。

注意

strcmp()函数比较的是字符串，不是字符，所以其参数应该是字符串（如"apples"和"A"），而不是字符（如'A'）。但是，char 类型实际上是整数类型，所以可以使用关系运算符来比较字符。假设 word 是储存在 char 类型数组中的字符串，ch 是 char 类型的变量，下面的语句都有效：

```
if (strcmp(word, "quit") == 0) // 使用 strcmp()比较字符串
    puts("Bye!");
if (ch == 'q') // 使用 == 比较字符
    puts("Bye!");
```

尽管如此，不要使用 ch 或'q'作为 strcmp()的参数。

程序清单 11.23 用 strcmp()函数检查程序是否要停止读取输入。

程序清单 11.23　quit_chk.c 程序

```
/* quit_chk.c -- 某程序的开始部分 */
#include <stdio.h>
#include <string.h>
```

```
#define SIZE 80
#define LIM 10
#define STOP "quit"
char * s_gets(char * st, int n);

int main(void)
{
    char input[LIM][SIZE];
    int ct = 0;

    printf("Enter up to %d lines (type quit to quit):\n", LIM);
    while (ct < LIM && s_gets(input[ct], SIZE) != NULL &&
                strcmp(input[ct], STOP) != 0)
    {
        ct++;
    }
    printf("%d strings entered\n", ct);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

该程序在读到 EOF 字符（这种情况下 s_gets()返回 NULL）、用户输入 quit 或输入项达到 LIM 时退出。

顺带一提，有时输入空行（即，只按下 Enter 键或 Return 键）表示结束输入更方便。为实现这一功能，只需修改一下 while 循环的条件即可：

```
while (ct < LIM && s_gets(input[ct], SIZE) != NULL&& input[ct][0] != '\0')
```

这里，input[ct]是刚输入的字符串，input[ct][0]是该字符串的第 1 个字符。如果用户输入空行，s_gets()便会把该行第 1 个字符（换行符）替换成空字符。所以，下面的表达式用于检测空行：

```
input[ct][0] != '\0'
```

2. strncmp()函数

strcmp()函数比较字符串中的字符，直到发现不同的字符为止，这一过程可能会持续到字符串的末尾。而 strncmp()函数在比较两个字符串时，可以比较到字符不同的地方，也可以只比较第 3 个参数指定的字

符数。例如，要查找以"astro"开头的字符串，可以限定函数只查找这 5 个字符。程序清单 11.24 演示了该函数的用法。

程序清单 11.24 starsrch.c **程序**

```
/* starsrch.c -- 使用 strncmp() */
#include <stdio.h>
#include <string.h>
#define LISTSIZE 6
int main()
{
    const char * list[LISTSIZE] =
    {
        "astronomy", "astounding",
        "astrophysics", "ostracize",
        "asterism", "astrophobia"
    };
    int count = 0;
    int i;

    for (i = 0; i < LISTSIZE; i++)
        if (strncmp(list[i], "astro", 5) == 0)
        {
            printf("Found: %s\n", list[i]);
            count++;
        }
    printf("The list contained %d words beginning"
           " with astro.\n", count);

    return 0;
}
```

下面是该程序的输出：

```
Found: astronomy
Found: astrophysics
Found: astrophobia
The list contained 3 words beginning with astro.
```

11.5.5　strcpy()和 strncpy()函数

前面提到过，如果 pts1 和 pts2 都是指向字符串的指针，那么下面语句拷贝的是字符串的地址而不是字符串本身：

```
pts2 = pts1;
```

如果希望拷贝整个字符串，要使用 strcpy()函数。程序清单 11.25 要求用户输入以 q 开头的单词。该程序把输入拷贝至一个临时数组中，如果第 1 个字母是 q，程序调用 strcpy()把整个字符串从临时数组拷贝至目标数组中。strcpy()函数相当于字符串赋值运算符。

程序清单 11.25 copy1.c **程序**

```
/* copy1.c -- 演示 strcpy() */
#include <stdio.h>
#include <string.h>  // strcpy()的原型在该头文件中
#define SIZE 40
#define LIM 5
```

```
char * s_gets(char * st, int n);

int main(void)
{
    char qwords[LIM][SIZE];
    char temp[SIZE];
    int i = 0;

    printf("Enter %d words beginning with q:\n", LIM);
    while (i < LIM && s_gets(temp, SIZE))
    {
        if (temp[0] != 'q')
            printf("%s doesn't begin with q!\n", temp);
        else
        {
            strcpy(qwords[i], temp);
            i++;
        }
    }
    puts("Here are the words accepted:");
    for (i = 0; i < LIM; i++)
        puts(qwords[i]);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                 continue;
    }
    return ret_val;
}
```

下面是该程序的运行示例：

```
Enter 5 words beginning with q:
quackery
quasar
quilt
quotient
no more
no more doesn't begin with q!
quiz
Here are the words accepted:
```

```
quackery
quasar
quilt
quotient
quiz
```

注意，只有在输入以 q 开头的单词后才会递增计数器 i，而且该程序通过比较字符进行判断：

```
if (temp[0] != 'q')
```

这行代码的意思是：temp 中的第 1 个字符是否是 q？当然，也可以通过比较字符串进行判断：

```
if (strncmp(temp, "q", 1) != 0)
```

这行代码的意思是：temp 字符串和"q"的第 1 个元素是否相等？

请注意，strcpy()第 2 个参数（temp）指向的字符串被拷贝至第 1 个参数（qword[i]）指向的数组中。拷贝出来的字符串被称为目标字符串，最初的字符串被称为源字符串。参考赋值表达式语句，很容易记住 strcpy()参数的顺序，即第 1 个是目标字符串，第 2 个是源字符串。

```
char target[20];
int x;
x = 50;                     /* 数字赋值*/
strcpy(target, "Hi ho!");   /* 字符串赋值*/
target = "So long";         /* 语法错误 */
```

程序员有责任确保目标数组有足够的空间容纳源字符串的副本。下面的代码有点问题：

```
char * str;
strcpy(str, "The C of Tranquility");    // 有问题
```

strcpy()把"The C of Tranquility"拷贝至 str 指向的地址上，但是 str 未被初始化，所以该字符串可能被拷贝到任意的地方！

总之，strcpy()接受两个字符串指针作为参数，可以把指向源字符串的第 2 个指针声明为指针、数组名或字符串常量；而指向源字符串副本的第 1 个指针应指向一个数据对象（如，数组），且该对象有足够的空间储存源字符串的副本。记住，声明数组将分配储存数据的空间，而声明指针只分配储存一个地址的空间。

1. strcpy()的其他属性

strcpy()函数还有两个有用的属性。第一，strcpy()的返回类型是 char *，该函数返回的是第 1 个参数的值，即一个字符的地址。第二，第 1 个参数不必指向数组的开始。这个属性可用于拷贝数组的一部分。程序清单 11.26 演示了该函数的这两个属性。

程序清单 11.26　copy2.c 程序

```
/* copy2.c -- 使用 strcpy() */
#include <stdio.h>
#include <string.h>    // 提供 strcpy()的函数原型
#define WORDS  "beast"
#define SIZE 40

int main(void)
{
    const char * orig = WORDS;
    char copy[SIZE] = "Be the best that you can be.";
    char * ps;

    puts(orig);
```

```
    puts(copy);
    ps = strcpy(copy + 7, orig);
    puts(copy);
    puts(ps);

    return 0;
}
```

下面是该程序的输出：

```
beast
Be the best that you can be.
Be the beast
beast
```

注意，strcpy()把源字符串中的空字符也拷贝在内。在该例中，空字符覆盖了 copy 数组中 that 的第 1 个 t（见图 11.5）。注意，由于第 1 个参数是 copy + 7，所以 ps 指向 copy 中的第 8 个元素（下标为 7）。因此 puts(ps)从该处开始打印字符串。

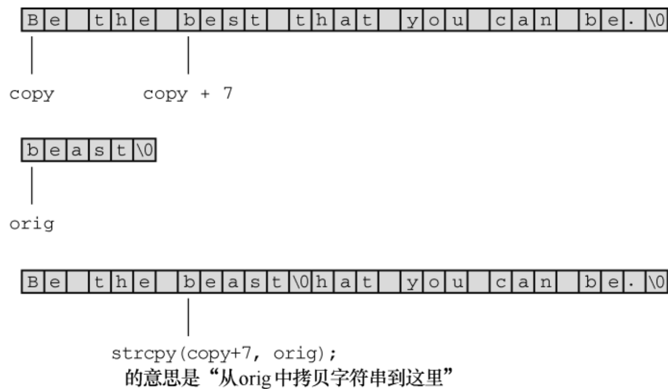


图 11.5　使用指针 strcpy()函数

2. 更谨慎的选择：strncpy()

strcpy()和 strcat()都有同样的问题，它们都不能检查目标空间是否能容纳源字符串的副本。拷贝字符串用 strncpy()更安全，该函数的第 3 个参数指明可拷贝的最大字符数。程序清单 11.27 用 strncpy()代替程序清单 11.25 中的 strcpy()。为了演示目标空间装不下源字符串的副本会发生什么情况，该程序使用了一个相当小的目标字符串（共 7 个元素，包含 6 个字符）。

程序清单 11.27　copy3.c 程序

```
/* copy3.c -- 使用strncpy() */
#include <stdio.h>
#include <string.h>    /* 提供strncpy()的函数原型*/
#define SIZE 40
#define TARGSIZE 7
#define LIM 5
char * s_gets(char * st, int n);

int main(void)
{
    char qwords[LIM][TARGSIZE];
    char temp[SIZE];
```

```
    int i = 0;

    printf("Enter %d words beginning with q:\n", LIM);
    while (i < LIM && s_gets(temp, SIZE))
    {
        if (temp[0] != 'q')
            printf("%s doesn't begin with q!\n", temp);
        else
        {
            strncpy(qwords[i], temp, TARGSIZE - 1);
            qwords[i][TARGSIZE - 1] = '\0';
            i++;
        }
    }
    puts("Here are the words accepted:");
    for (i = 0; i < LIM; i++)
        puts(qwords[i]);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                  continue;
    }
    return ret_val;
}
```

下面是该程序的运行示例：

```
Enter 5 words beginning with q:
quack
quadratic
quisling
quota
quagga
Here are the words accepted:
quack
quadra
quisli
quota
quagga
```

strncpy(target, source, n)把 source 中的 n 个字符或空字符之前的字符（先满足哪个条件就

拷贝到何处）拷贝至 target 中。因此，如果 source 中的字符数小于 n，则拷贝整个字符串，包括空字符。但是，strncpy()拷贝字符串的长度不会超过 n，如果拷贝到第 n 个字符时还未拷贝完整个源字符串，就不会拷贝空字符。所以，拷贝的副本中不一定有空字符。鉴于此，该程序把 n 设置为比目标数组大小少 1（TARGSIZE-1），然后把数组最后一个元素设置为空字符：

```
strncpy(qwords[i], temp, TARGSIZE - 1);
qwords[i][TARGSIZE - 1] = '\0';
```

这样做确保储存的是一个字符串。如果目标空间能容纳源字符串的副本，那么从源字符串拷贝的空字符便是该副本的结尾；如果目标空间装不下副本，则把副本最后一个元素设置为空字符。

11.5.6　sprintf()函数

sprintf()函数声明在 stdio.h 中，而不是在 string.h 中。该函数和 printf()类似，但是它是把数据写入字符串，而不是打印在显示器上。因此，该函数可以把多个元素组合成一个字符串。sprintf()的第 1 个参数是目标字符串的地址。其余参数和 printf()相同，即格式字符串和待写入项的列表。

程序清单 11.28 中的程序用 printf()把 3 个项（两个字符串和一个数字）组合成一个字符串。注意，sprintf()的用法和 printf()相同，只不过 sprintf()把组合后的字符串储存在数组 formal 中而不是显示在屏幕上。

程序清单 11.28　format.c 程序

```
/* format.c -- 格式化字符串 */
#include <stdio.h>
#define MAX 20
char * s_gets(char * st, int n);

int main(void)
{
    char first[MAX];
    char last[MAX];
    char formal[2 * MAX + 10];
    double prize;

    puts("Enter your first name:");
    s_gets(first, MAX);
    puts("Enter your last name:");
    s_gets(last, MAX);
    puts("Enter your prize money:");
    scanf("%lf", &prize);
    sprintf(formal, "%s, %-19s: $%6.2f\n", last, first, prize);
    puts(formal);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
```

```
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

下面是该程序的运行示例：

```
Enter your first name:
Annie
Enter your last name:
von Wurstkasse
Enter your prize money:
25000
von Wurstkasse, Annie          : $25000.00
```

sprintf()函数获取输入，并将其格式化为标准形式，然后把格式化后的字符串储存在 formal 中。

11.5.7 其他字符串函数

ANSI C 库有 20 多个用于处理字符串的函数，下面总结了一些常用的函数。

- char *strcpy(char * restrict s1, const char * restrict s2);

  该函数把 s2 指向的字符串（包括空字符）拷贝至 s1 指向的位置，返回值是 s1。

- char *strncpy(char * restrict s1, const char * restrict s2, size_t n);

  该函数把 s2 指向的字符串拷贝至 s1 指向的位置，拷贝的字符数不超过 n，其返回值是 s1。该函数不会拷贝空字符后面的字符，如果源字符串的字符少于 n 个，目标字符串就以拷贝的空字符结尾；如果源字符串有 n 个或超过 n 个字符，就不拷贝空字符。

- char *strcat(char * restrict s1, const char * restrict s2);

  该函数把 s2 指向的字符串拷贝至 s1 指向的字符串末尾。s2 字符串的第 1 个字符将覆盖 s1 字符串末尾的空字符。该函数返回 s1。

- char *strncat(char * restrict s1, const char * restrict s2, size_t n);

  该函数把 s2 字符串中的 n 个字符拷贝至 s1 字符串末尾。s2 字符串的第 1 个字符将覆盖 s1 字符串末尾的空字符。不会拷贝 s2 字符串中空字符和其后的字符，并在拷贝字符的末尾添加一个空字符。该函数返回 s1。

- int strcmp(const char * s1, const char * s2);

  如果 s1 字符串在机器排序序列中位于 s2 字符串的后面，该函数返回一个正数；如果两个字符串相等，则返回 0；如果 s1 字符串在机器排序序列中位于 s2 字符串的前面，则返回一个负数。

- int strncmp(const char * s1, const char * s2, size_t n);

  该函数的作用和 strcmp()类似，不同的是，该函数在比较 n 个字符后或遇到第 1 个空字符时停止比较。

- char *strchr(const char * s, int c);

如果 s 字符串中包含 c 字符，该函数返回指向 s 字符串首位置的指针（末尾的空字符也是字符串的一部分，所以在查找范围内）；如果在字符串 s 中未找到 c 字符，该函数则返回空指针。

- `char *strpbrk(const char * s1, const char * s2);`

  如果 s1 字符中包含 s2 字符串中的任意字符，该函数返回指向 s1 字符串首位置的指针；如果在 s1 字符串中未找到任何 s2 字符串中的字符，则返回空字符。

- `char *strrchr(const char * s, int c);`

  该函数返回 s 字符串中 c 字符的最后一次出现的位置（末尾的空字符也是字符串的一部分，所以在查找范围内）。如果未找到 c 字符，则返回空指针。

- `char *strstr(const char * s1, const char * s2);`

  该函数返回指向 s1 字符串中 s2 字符串出现的首位置。如果在 s1 中没有找到 s2，则返回空指针。

- `size_t strlen(const char * s);`

  该函数返回 s 字符串中的字符数，不包括末尾的空字符。

请注意，那些使用 const 关键字的函数原型表明，函数不会更改字符串。例如，下面的函数原型：

```
char *strcpy(char * restrict s1, const char * restrict s2);
```

表明不能更改 s2 指向的字符串，至少不能在 strcpy()函数中更改。但是可以更改 s1 指向的字符串。这样做很合理，因为 s1 是目标字符串，要改变，而 s2 是源字符串，不能更改。

关键字 restrict 将在第 12 章中介绍，该关键字限制了函数参数的用法。例如，不能把字符串拷贝给本身。

第 5 章中讨论过，size_t 类型是 sizeof 运算符返回的类型。C 规定 sizeof 运算符返回一个整数类型，但是并未指定是哪种整数类型，所以 size_t 在一个系统中可以是 unsigned int，而在另一个系统中可以是 unsigned long。string.h 头文件针对特定系统定义了 size_t，或者参考其他有 size_t 定义的头文件。

前面提到过，参考资料 V 中列出了 string.h 系列的所有函数。除提供 ANSI 标准要求的函数外，许多实现还提供一些其他函数。应查看你所使用的 C 实现文档，了解可以使用哪些函数。

我们来看一下其中一个函数的简单用法。前面学过的 fgets()读入一行输入时，在目标字符串的末尾添加换行符。我们自定义的 s_gets()函数通过 while 循环检测换行符。其实，这里可以用 strchr()代替 s_gets()。首先，使用 strchr()查找换行符（如果有的话）。如果该函数发现了换行符，将返回该换行符的地址，然后便可用空字符替换该位置上的换行符：

```
char line[80];
char * find;

fgets(line, 80, stdin);
find = strchr(line, '\n'); // 查找换行符
if (find)                  // 如果没找到换行符，返回NULL
    *find = '\0';          // 把该处的字符替换为空字符
```

如果 strchr()未找到换行符，fgets()在达到行末尾之前就达到了它能读取的最大字符数。可以像在 s_gets()中那样，给 if 添加一个 else 来处理这种情况。

接下来，我们看一个处理字符串的完整程序。

11.6　字符串示例：字符串排序

我们来处理一个按字母表顺序排序字符串的实际问题。准备名单表、创建索引和许多其他情况下都会用到字符串排序。该程序主要是用 strcmp()函数来确定两个字符串的顺序。一般的做法是读取字符串函数、排序字符串并打印出来。之前，我们设计了一个读取字符串的方案，该程序就用到这个方案。打印字符串没问题。程序使用标准的排序算法，稍后解释。我们使用了一个小技巧，看看读者是否能明白。程序清单 11.29 演示了这个程序。

程序清单 11.29　sort_str.c 程序

```
/* sort_str.c -- 读入字符串，并排序字符串 */
#include <stdio.h>
#include <string.h>
#define SIZE 81         /* 限制字符串长度，包括 \0 */
#define LIM 20          /* 可读入的最多行数 */
#define HALT ""         /* 空字符串停止输入 */
void stsrt(char *strings [], int num);   /* 字符串排序函数 */
char * s_gets(char * st, int n);

int main(void)
{
    char input[LIM][SIZE];      /* 储存输入的数组         */
    char *ptstr[LIM];           /* 内含指针变量的数组     */
    int ct = 0;                 /* 输入计数               */
    int k;                      /* 输出计数               */

    printf("Input up to %d lines, and I will sort them.\n", LIM);
    printf("To stop, press the Enter key at a line's start.\n");
    while (ct < LIM && s_gets(input[ct], SIZE) != NULL
            && input[ct][0] != '\0')
    {
        ptstr[ct] = input[ct];  /* 设置指针指向字符串    */
        ct++;
    }
    stsrt(ptstr, ct);           /* 字符串排序函数         */
    puts("\nHere's the sorted list:\n");
    for (k = 0; k < ct; k++)
        puts(ptstr[k]);         /* 排序后的指针           */

    return 0;
}

/* 字符串-指针-排序函数 */
void stsrt(char *strings [], int num)
{
    char *temp;
    int top, seek;

    for (top = 0; top < num - 1; top++)
        for (seek = top + 1; seek < num; seek++)
            if (strcmp(strings[top], strings[seek]) > 0)
```

```
            {
                temp = strings[top];
                strings[top] = strings[seek];
                strings[seek] = temp;
            }
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                  continue;
    }
    return ret_val;
}
```

我们用一首童谣来测试该程序：

```
Input up to 20 lines, and I will sort them.
To stop, press the Enter key at a line's start.
O that I was where I would be,
Then would I be where I am not;
But where I am I must be,
And where I would be I can not.

Here's the sorted list:

And where I would be I can not.
But where I am I must be,
O that I was where I would be,
Then would I be where I am not;
```

看来经过排序后，这首童谣的内容未受影响。

11.6.1　排序指针而非字符串

该程序的巧妙之处在于排序的是指向字符串的指针，而不是字符串本身。我们来分析一下具体怎么做。最初，ptrst[0]被设置为 input[0]，ptrst[1]被设置为 input[1]，以此类推。这意味着指针 ptrst[i]指向数组 input[i]的首字符。每个 input[i]都是一个内含 81 个元素的数组，每个 ptrst[i]都是一个单独的变量。排序过程把 ptrst 重新排列，并未改变 input。例如，如果按字母顺序 input[1]在 intput[0]前面，程序便交换指向它们的指针（即 ptrst[0]指向 input[1]的开始，而 ptrst[1]指向 input[0]的开始）。这样做比用 strcpy()交换两个 input 字符串的内容简单得多，而且还保留了 input 数组中的原始顺序。图 11.6 从另一个视角演示了这一过程。

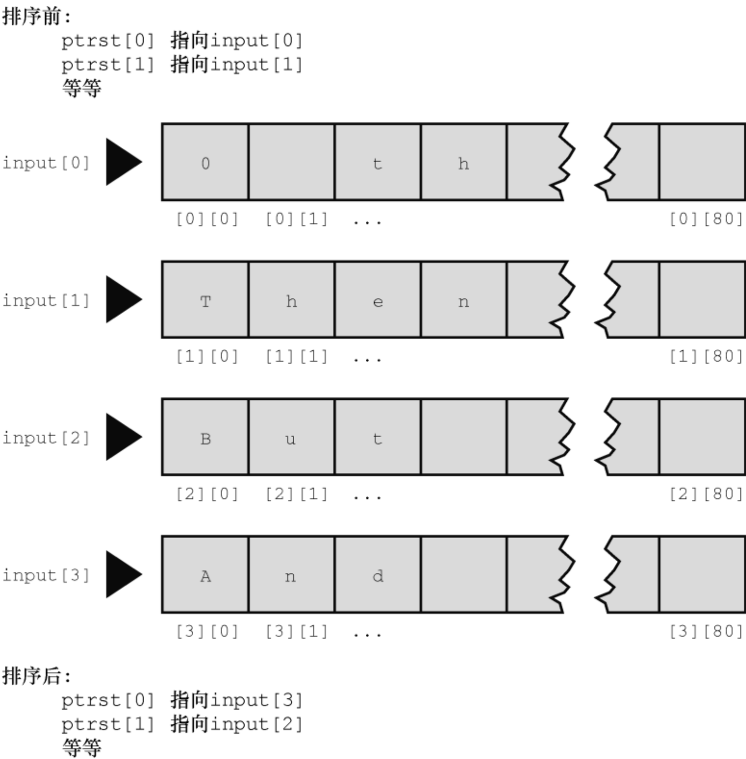


图 11.6　排序字符串指针

11.6.2　选择排序算法

我们采用*选择排序算法*（*selection sort algorithm*）来排序指针。具体做法是，利用 `for` 循环依次把每个元素与首元素比较。如果待比较的元素在当前首元素的前面，则交换两者。循环结束时，首元素包含的指针指向机器排序序列最靠前的字符串。然后外层 `for` 循环重复这一过程，这次从 `input` 的第 2 个元素开始。当内层循环执行完毕时，`ptrst` 中的第 2 个元素指向排在第 2 的字符串。这一过程持续到所有元素都已排序完毕。

现在来进一步分析选择排序的过程。下面是排序过程的伪代码：

```
for n = 首元素至 n = 倒数第 2 个元素,
    找出剩余元素中的最大值，并将其放在第 n 个元素中
```

具体过程如下。首先，从 `n = 0` 开始，遍历整个数组找出最大值元素，那该元素与第 1 个元素交换；然后设置 `n = 1`，遍历除第 1 个元素以外的其他元素，在其余元素中找出最大值元素，把该元素与第 2 个元素交换；重复这一过程直至倒数第 2 个元素为止。现在只剩下两个元素。比较这两个元素，把较大者放在倒数第 2 的位置。这样，数组中的最小元素就在最后的位置上。

这看起来用 `for` 循环就能完成任务，但是我们还要更详细地分析“查找和放置”的过程。在剩余项中查找最大值的方法是，比较数组剩余元素的第 1 个元素和第 2 个元素。如果第 2 个元素比第 1 个元素大，交换两者。现在比较数组剩余元素的第 1 个元素和第 3 个元素，如果第 3 个元素比较大，交换两者。每次交换都把较大的元素移至顶部。继续这一过程直到比较第 1 个元素和最后一个元素。比较完毕后，最大值元素现在是剩余数组的首元素。已经排出了该数组的首元素，但是其他元素还是一团糟。下面是排序过程的伪代码：

```
for n - 第 2 个元素至最后一个元素，
    比较第 n 个元素与第 1 个元素，如果第 n 个元素更大，交换这两个元素的值
```

看上去用一个 for 循环也能搞定。只不过要把它嵌套在刚才的 for 循环中。外层循环指明正在处理数组的哪一个元素，内层循环找出应储存在该元素的值。把这两部分伪代码结合起来，翻译成 C 代码，就得到了程序清单 11.29 中的 stsrt()函数。顺带一提，C 库中有一个更高级的排序函数：qsort()。该函数使用一个指向函数的指针进行排序比较。第 16 章将给出该函数的用法示例。

11.7　ctype.h 字符函数和字符串

第 7 章中介绍了 ctype.h 系列与字符相关的函数。虽然这些函数不能处理整个字符串，但是可以处理字符串中的字符。例如，程序清单 11.30 中定义的 ToUpper()函数，利用 toupper()函数处理字符串中的每个字符，把整个字符串转换成大写；定义的 PunctCount()函数，利用 ispunct()统计字符串中的标点符号个数。另外，该程序使用 strchr()处理 fgets()读入字符串的换行符（如果有的话）。

程序清单 11.30　mod_str.c 程序

```
/* mod_str.c -- 修改字符串 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LIMIT 81
void ToUpper(char *);
int PunctCount(const char *);

int main(void)
{
    char line[LIMIT];
    char * find;

    puts("Please enter a line:");
    fgets(line, LIMIT, stdin);
    find = strchr(line, '\n'); // 查找换行符
    if (find)              // 如果地址不是 NULL,
        *find = '\0';          // 用空字符替换
    ToUpper(line);
    puts(line);
    printf("That line has %d punctuation characters.\n", PunctCount(line));

    return 0;
}

void ToUpper(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}
```

```
int PunctCount(const char * str)
{
    int ct = 0;
    while (*str)
    {
        if (ispunct(*str))
            ct++;
        str++;
    }

    return ct;
}
```

while (*str)循环处理 str 指向的字符串中的每个字符，直至遇到空字符。此时*str 的值为 0（空字符的编码值为 0），即循环条件为假，循环结束。下面是该程序的运行示例：

```
Please enter a line:
Me? You talkin' to me? Get outta here!
ME? YOU TALKIN' TO ME? GET OUTTA HERE!
That line has 4 punctuation characters.
```

ToUpper()函数利用 toupper()处理字符串中的每个字符（由于 C 区分大小写，所以这是两个不同的函数名）。根据 ANSI C 中的定义，toupper()函数只改变小写字符。但是一些很旧的 C 实现不会自动检查大小写，所以以前的代码通常会这样写：

```
if (islower(*str)) /* ANSI C之前的做法 -- 在转换大小写之前先检查 */
    *str = toupper(*str);
```

顺带一提，ctype.h 中的函数通常作为宏（*macro*）来实现。这些 C 预处理器宏的作用很像函数，但是两者有一些重要的区别。我们在第 16 章再讨论关于宏的内容。

该程序使用 fgets()和 strchr()组合，读取一行输入并把换行符替换成空字符。这种方法与使用 s_gets()的区别是：s_gets()会处理输入行剩余字符（如果有的话），为下一次输入做好准备。而本例只有一条输入语句，就没必要进行多余的步骤。

11.8　命令行参数

在图形界面普及之前都使用命令行界面。DOS 和 UNIX 就是例子。Linux 终端提供类 UNIX 命令行环境。命令行（*command line*）是在命令行环境中，用户为运行程序输入命令的行。假设一个文件中有一个名为 fuss 的程序。在 UNIX 环境中运行该程序的命令行是：

```
$ fuss
```

或者在 Windows 命令提示模式下是：

```
C> fuss
```

命令行参数（*command-line argument*）是同一行的附加项。如下例：

```
$ fuss -r Ginger
```

一个 C 程序可以读取并使用这些附加项（见图 11.7）。

程序清单 11.27 是一个典型的例子，该程序通过 main()的参数读取这些附加项。

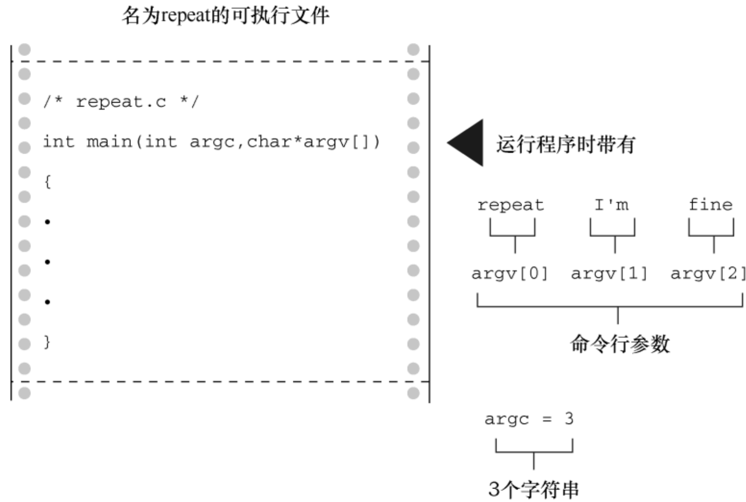


图 11.7　命令行参数

程序清单 11.31　`repeat.c` **程序**

```
/* repeat.c -- 带参数的 main() */
#include <stdio.h>
int main(int argc, char *argv [])
{
    int count;

    printf("The command line has %d arguments:\n", argc - 1);
    for (count = 1; count < argc; count++)
        printf("%d: %s\n", count, argv[count]);
    printf("\n");

    return 0;
}
```

把该程序编译为可执行文件 `repeat`。下面是通过命令行运行该程序后的输出：

```
C>repeat Resistance is futile
The command line has 3 arguments:
1: Resistance
2: is
3: futile
```

由此可见该程序为何名为 `repeat`。下面我们解释一下它的运行原理。

C 编译器允许 `main()`没有参数或者有两个参数（一些实现允许 `main()`有更多参数，属于对标准的扩展）。`main()`有两个参数时，第 1 个参数是命令行中的字符串数量。过去，这个 `int` 类型的参数被称为 `argc`（表示参数计数 *(argument count)*）。系统用空格表示一个字符串的结束和下一个字符串的开始。因此，上面的 `repeat` 示例中包括命令名共有 4 个字符串，其中后 3 个供 `repeat` 使用。该程序把命令行字符串储存在内存中，并把每个字符串的地址储存在指针数组中。而该数组的地址则被储存在 `main()`的第 2 个参数中。按照惯例，这个指向指针的指针称为 `argv`（表示参数值[*argument value*]）。如果系统允许（一些操作系统不允许这样），就把程序本身的名称赋给 `argv[0]`，然后把随后的第 1 个字符串赋给 `argv[1]`，以此类推。在我们的例子中，有下面的关系：

argv[0] 指向 repeat （对大部分系统而言）

argv[1] 指向 Resistance

argv[2] 指向 is

argv[3] 指向 futile

程序清单 11.31 的程序通过一个 for 循环依次打印每个字符串。printf()中的%s 转换说明表明，要提供一个字符串的地址作为参数，而指针数组中的每个元素（argv[0]、argv[1]等）都是这样的地址。

main()中的形参形式与其他带形参的函数相同。许多程序员用不同的形式声明 argv：

```
int main(int argc, char **argv)
```

char **argv 与 char *argv[]等价。也就是说，argv 是一个指向指针的指针，它所指向的指针指向 char。因此，即使在原始定义中，argv 也是指向指针（该指针指向 char）的指针。两种形式都可以使用，但我们认为第 1 种形式更清楚地表明 argv 表示一系列字符串。

顺带一提，许多环境（包括 UNIX 和 DOS）都允许用双引号把多个单词括起来形成一个参数。例如：

```
repeat "I am hungry" now
```

这行命令把字符串"I am hungry"赋给 argv[1]，把"now"赋给 argv[2]。

11.8.1 集成环境中的命令行参数

Windows 集成环境（如 Xcode、Microsoft Visual C++和 Embarcadero C++ Builder）都不用命令行运行程序。有些环境中有项目对话框，为特定项目指定命令行参数。其他环境中，可以在 IDE 中编译程序，然后打开 MS-DOS 窗口在命令行模式中运行程序。但是，如果你的系统有一个运行命令行的编译器（如 GCC）会更简单。

11.8.2 Macintosh 中的命令行参数

如果使用 Xcode 4.6（或类似的版本），可以在 Product 菜单中选择 Scheme 选项来提供命令行参数，编辑 Scheme，运行。然后选择 Argument 标签，在 Launch 的 Arguments Pass 中输入参数。

或者进入 Mac 的 Terminal 模式和 UNIX 的命令行环境。然后，可以找到程序可执行代码的目录（UNIX 的文件夹），或者下载命令行工具，使用 gcc 或 clang 编译程序。

11.9 把字符串转换为数字

数字既能以字符串形式储存，也能以数值形式储存。把数字储存为字符串就是储存数字字符。例如，数字 213 以'2'、'1'、'3'、'\0'的形式被储存在字符串数组中。以数值形式储存 213，储存的是 int 类型的值。

C 要求用数值形式进行数值运算（如，加法和比较）。但是在屏幕上显示数字则要求字符串形式，因为屏幕显示的是字符。printf()和 sprintf()函数，通过%d 和其他转换说明，把数字从数值形式转换为字符串形式，scanf()可以把输入字符串转换为数值形式。C 还有一些函数专门用于把字符串形式转换成数值形式。

假设你编写的程序需要使用数值命令形参，但是命令形参数被读取为字符串。因此，要使用数值必须先把字符串转换为数字。如果需要整数，可以使用 atoi()函数（用于把字母数字转换成整数），该函数接受一个字符串作为参数，返回相应的整数值。程序清单 11.32 中的程序示例演示了该函数的用法。

程序清单 11.32　hello.c 程序

```
/* hello.c -- 把命令行参数转换为数字 */
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv [])
{
    int i, times;

    if (argc < 2 || (times = atoi(argv[1])) < 1)
        printf("Usage: %s positive-number\n", argv[0]);
    else
        for (i = 0; i < times; i++)
            puts("Hello, good looking!");
    return 0;
}
```

该程序的运行示例：

```
$ hello 3
Hello, good looking!
Hello, good looking!
Hello, good looking!
```

$是 UNIX 和 Linux 的提示符（一些 UNIX 系统使用%）。命令行参数 3 被储存为字符串 3\0。atoi()函数把该字符串转换为整数值 3，然后该值被赋给 times。该值确定了执行 for 循环的次数。

如果运行该程序时没有提供命令行参数，那么 argc < 2 为真，程序给出一条提示信息后结束。如果 times 为 0 或负数，情况也是如此。C 语言逻辑运算符的求值顺序保证了如果 argc < 2，就不会对 atoi(argv[1])求值。

如果字符串仅以整数开头，atio()函数也能处理，它只把开头的整数转换为字符。例如，atoi("42regular")将返回整数 42。如果在命令行输入 hello what 会怎样？在我们所用的 C 实现中，如果命令行参数不是数字，atoi()函数返回 0。然而 C 标准规定，这种情况下的行为是未定义的。因此，使用有错误检测功能的 strtol()函数（马上介绍）会更安全。

该程序中包含了 stdlib.h 头文件，因为从 ANSI C 开始，该头文件中包含了 atoi()函数的原型。除此之外，还包含了 atof()和 atol()函数的原型。atof()函数把字符串转换成 double 类型的值，atol()函数把字符串转换成 long 类型的值。atof()和 atol()的工作原理和 atoi()类似，因此它们分别返回 double 类型和 long 类型。

ANSI C 还提供一套更智能的函数：strtol()把字符串转换成 long 类型的值，strtoul()把字符串转换成 unsigned long 类型的值，strtod()把字符串转换成 double 类型的值。这些函数的智能之处在于识别和报告字符串中的首字符是否是数字。而且，strtol()和 strtoul()还可以指定数字的进制。

下面的程序示例中涉及 strtol()函数，其原型如下：

```
long strtol(const char * restrict nptr, char ** restrict endptr, int base);
```

这里，nptr 是指向待转换字符串的指针，endptr 是一个指针的地址，该指针被设置为标识输入数字结束字符的地址，base 表示以什么进制写入数字。程序清单 11.33 演示了该函数的用法。

程序清单 11.33　strcnvt.c 程序

```
/* strcnvt.c -- 使用 strtol()  */
```

```
#include <stdio.h>
#include <stdlib.h>
#define LIM 30
char * s_gets(char * st, int n);

int main()
{
    char number[LIM];
    char * end;
    long value;

    puts("Enter a number (empty line to quit):");
    while (s_gets(number, LIM) && number[0] != '\0')
    {
        value = strtol(number, &end, 10);  /* 十进制 */
        printf("base 10 input, base 10 output: %ld, stopped at %s (%d)\n",
                   value, end, *end);
        value = strtol(number, &end, 16);  /* 十六进制 */
        printf("base 16 input, base 10 output: %ld, stopped at %s (%d)\n",
                   value, end, *end);
        puts("Next number:");
    }
    puts("Bye!\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

下面是该程序的输出示例：

```
Enter a number (empty line to quit):
10
base 10 input, base 10 output: 10, stopped at (0)
base 16 input, base 10 output: 16, stopped at (0)
Next number:
10atom
base 10 input, base 10 output: 10, stopped at atom (97)
```

```
base 16 input, base 10 output: 266, stopped at tom (116)
Next number:

Bye!
```

首先注意，当 base 分别为 10 和 16 时，字符串"10"分别被转换成数字 10 和 16。还要注意，如果 end 指向一个字符，*end 就是一个字符。因此，第 1 次转换在读到空字符时结束，此时 end 指向空字符。打印 end 会显示一个空字符串，以%d 转换说明输出*end 显示的是空字符的 ASCII 码。

对于第 2 个输入的字符串，当 base 为 10 时，end 的值是'a'字符的地址。所以打印 end 显示的是字符串"atom"，打印*end 显示的是'a'字符的 ASCII 码。然而，当 base 为 16 时，'a'字符被识别为一个有效的十六进制数，strtol()函数把十六进制数 10a 转换成十进制数 266。

strtol()函数最多可以转换三十六进制，'a'~'z'字符都可用作数字。strtoul()函数与该函数类似，但是它把字符串转换成无符号值。strtod()函数只以十进制转换，因此它值需要两个参数。

许多实现使用 itoa()和 ftoa()函数分别把整数和浮点数转换成字符串。但是这两个函数并不是 C 标准库的成员，可以用 sprintf()函数代替它们，因为 sprintf()的兼容性更好。

11.10　关键概念

许多程序都要处理文本数据。一个程序可能要求用户输入姓名、公司列表、地址、一种蕨类植物的学名、音乐剧的演员等。毕竟，我们用言语与现实世界互动，使用文本的例子不计其数。C 程序通过字符串的方式来处理它们。

字符串，无论是由字符数组、指针还是字符串常量标识，都储存为包含字符编码的一系列字节，并以空字符串结尾。C 提供库函数处理字符串，查找字符串并分析它们。尤其要牢记，应该使用 strcmp()来代替关系运算符，当比较字符串时，应该使用 strcpy()或 strncpy()代替赋值运算符把字符串赋给字符数组。

11.11　本章小结

C 字符串是一系列 char 类型的字符，以空字符（'\0'）结尾。字符串可以储存在字符数组中。字符串还可以用字符串常量来表示，里面都是字符，括在双引号中（空字符除外）。编译器提供空字符。因此，"joy"被储存为 4 个字符 j、o、y 和\0。strlen()函数可以统计字符串的长度，空字符不计算在内。

字符串常量也叫作字符串——字面量，可用于初始化字符数组。为了容纳末尾的空字符，数组大小应该至少比容纳的数组长度多 1。也可以用字符串常量初始化指向 char 的指针。

函数使用指向字符串首字符的指针来表示待处理的字符串。通常，对应的实际参数是数组名、指针变量或用双引号括起来的字符串。无论是哪种情况，传递的都是首字符的地址。一般而言，没必要传递字符串的长度，因为函数可以通过末尾的空字符确定字符串的结束。

fgets()函数获取一行输入，puts()和 fputs()函数显示一行输出。它们都是 stdio.h 头文件中的函数，用于代替已被弃用的 gets()。

C 库中有多个字符串处理函数。在 ANSI C 中，这些函数都声明在 string.h 文件中。C 库中还有许多字符处理函数，声明在 ctype.h 文件中。

给 main()函数提供两个合适的形式参数，可以让程序访问命令行参数。第 1 个参数通常是 int 类型的 argc，其值是命令行的单词数量。第 2 个参数通常是一个指向数组的指针 argv，数组内含指向 char 的指针。每个指向 char 的指针都指向一个命令行参数字符串，argv[0]指向命令名称，argv[1]指向第 1 个命令行参数，以此类推。

atoi()、atol()和 atof()函数把字符串形式的数字分别转换成 int、long 和 double 类型的数字。strtol()、strtoul()和 strtod()函数把字符串形式的数字分别转换成 long、unsigned long 和 double 类型的数字。

11.12 复习题

复习题的参考答案在附录 A 中。

1. 下面字符串的声明有什么问题？

```
int main(void)
{
    char name[] = {'F', 'e', 's', 's' };
    ...
}
```

2. 下面的程序会打印什么？

```
#include <stdio.h>
int main(void)
{
    char note[] = "See you at the snack bar.";
    char *ptr;

    ptr = note;
    puts(ptr);
    puts(++ptr);
    note[7] = '\0';
    puts(note);
    puts(++ptr);
    return 0;
}
```

3. 下面的程序会打印什么？

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char food [] = "Yummy";
    char *ptr;

    ptr = food + strlen(food);
    while (--ptr >= food)
        puts(ptr);
    return 0;
}
```

4. 下面的程序会打印什么？

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char goldwyn[40] = "art of it all ";
    char samuel[40] = "I read p";
    const char * quote = "the way through.";

    strcat(goldwyn, quote);
    strcat(samuel, goldwyn);
```

```
    puts(samuel);
    return 0;
}
```

5. 下面的练习涉及字符串、循环、指针和递增指针。首先，假设定义了下面的函数：

```
#include <stdio.h>
char *pr(char *str)
{
    char *pc;

    pc = str;
    while (*pc)
        putchar(*pc++);
    do {
        putchar(*--pc);
       } while (pc - str);
    return (pc);
}
```

考虑下面的函数调用：

```
x = pr("Ho Ho Ho!");
```

a. 将打印什么？

b. x 是什么类型？

c. x 的值是什么？

d. 表达式*--pc 是什么意思？与--*pc 有何不同？

e. 如果用*--pc 替换--*pc，会打印什么？

f. 两个 while 循环用来测试什么？

g. 如果 pr()函数的参数是空字符串，会怎样？

h. 必须在主调函数中做什么，才能让 pr()函数正常运行？

6. 假设有如下声明：

```
char sign = '$';
```

sign 占用多少字节的内存？'$'占用多少字节的内存？"$"占用多少字节的内存？

7. 下面的程序会打印出什么？

```
#include <stdio.h>
#include <string.h>
#define M1 "How are ya, sweetie? "
char M2[40] = "Beat the clock.";
char * M3 = "chat";
int main(void)
{
    char words[80];
    printf(M1);
    puts(M1);
    puts(M2);
    puts(M2 + 1);
    strcpy(words, M2);
    strcat(words, " Win a toy.");
    puts(words);
    words[4] = '\0';
    puts(words);
```

```
    while (*M3)
        puts(M3++);
    puts(--M3);
    puts(--M3);
    M3 = M1;
    puts(M3);
    return 0;
}
```

8. 下面的程序会打印出什么？

```
#include <stdio.h>
int main(void)
{
    char str1 [] = "gawsie";
    char str2 [] = "bletonism";
    char *ps;
    int i = 0;
    for (ps = str1; *ps != '\0'; ps++) {
        if (*ps == 'a' || *ps == 'e')
            putchar(*ps);
        else
            (*ps)--;
        putchar(*ps);
        }
    putchar('\n');
    while (str2[i] != '\0') {
        printf("%c", i % 3 ? str2[i] : '*');
        ++i;
        }
    return 0;
}
```

9. 本章定义的 s_gets()函数，用指针表示法代替数组表示法便可减少一个变量 i。请改写该函数。

10. strlen()函数接受一个指向字符串的指针作为参数，并返回该字符串的长度。请编写一个这样的函数。

11. 本章定义的 s_gets()函数，可以用 strchr()函数代替其中的 while 循环来查找换行符。请改写该函数。

12. 设计一个函数，接受一个指向字符串的指针，返回指向该字符串第 1 个空格字符的指针，或如果未找到空格字符，则返回空指针。

13. 重写程序清单 11.21，使用 ctype.h 头文件中的函数，以便无论用户选择大写还是小写，该程序都能正确识别答案。

11.13 编程练习

1. 设计并测试一个函数，从输入中获取下 n 个字符（包括空白、制表符、换行符），把结果储存在一个数组里，它的地址被传递作为一个参数。

2. 修改并编程练习 1 的函数，在 n 个字符后停止，或在读到第 1 个空白、制表符或换行符时停止，哪个先遇到哪个停止。不能只使用 scanf()。

3. 设计并测试一个函数，从一行输入中把一个单词读入一个数组中，并丢弃输入行中的其余字符。该函数应该跳过第 1 个非空白字符前面的所有空白。将一个单词定义为没有空白、制表符或换行符的

字符序列。

4. 设计并测试一个函数，它类似编程练习 3 的描述，只不过它接受第 2 个参数指明可读取的最大字符数。

5. 设计并测试一个函数，搜索第 1 个函数形参指定的字符串，在其中查找第 2 个函数形参指定的字符首次出现的位置。如果成功，该函数返指向该字符的指针，如果在字符串中未找到指定字符，则返回空指针（该函数的功能与 strchr()函数相同）。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。

6. 编写一个名为 is_within()的函数，接受一个字符和一个指向字符串的指针作为两个函数形参。如果指定字符在字符串中，该函数返回一个非零值（即为真）。否则，返回 0（即为假）。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。

7. strncpy(s1, s2, n)函数把 s2 中的 n 个字符拷贝至 s1 中，截断 s2，或者有必要的话在末尾添加空字符。如果 s2 的长度是 n 或多于 n，目标字符串不能以空字符结尾。该函数返回 s1。自己编写一个这样的函数，名为 mystrncpy()。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。

8. 编写一个名为 string_in()的函数，接受两个指向字符串的指针作为参数。如果第 2 个字符串中包含第 1 个字符串，该函数将返回第 1 个字符串开始的地址。例如，string_in("hats", "at")将返回 hats 中 a 的地址。否则，该函数返回空指针。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。

9. 编写一个函数，把字符串中的内容用其反序字符串代替。在一个完整的程序中测试该函数，使用一个循环给函数提供输入值。

10. 编写一个函数接受一个字符串作为参数，并删除字符串中的空格。在一个程序中测试该函数，使用循环读取输入行，直到用户输入一行空行。该程序应该应用该函数只每个输入的字符串，并显示处理后的字符串。

11. 编写一个函数，读入 10 个字符串或者读到 EOF 时停止。该程序为用户提供一个有 5 个选项的菜单：打印源字符串列表、以 ASCII 中的顺序打印字符串、按长度递增顺序打印字符串、按字符串中第 1 个单词的长度打印字符串、退出。菜单可以循环显示，除非用户选择退出选项。当然，该程序要能真正完成菜单中各选项的功能。

12. 编写一个程序，读取输入，直至读到 EOF，报告读入的单词数、大写字母数、小写字母数、标点符号数和数字字符数。使用 ctype.h 头文件中的函数。

13. 编写一个程序，反序显示命令行参数的单词。例如，命令行参数是 see you later，该程序应打印 later you see。

14. 编写一个通过命令行运行的程序计算幂。第 1 个命令行参数是 double 类型的数，作为幂的底数，第 2 个参数是整数，作为幂的指数。

15. 使用字符分类函数实现 atoi()函数。如果输入的字符串不是纯数字，该函数返回 0。

16. 编写一个程序读取输入，直至读到文件结尾，然后把字符串打印出来。该程序识别和实现下面的命令行参数：

```
-p          按原样打印
-u          把输入全部转换成大写
-l          把输入全部转换成小写
```

如果没有命令行参数，则让程序像是使用了-p 参数那样运行。

第 12 章

存储类别、链接和内存管理

本章介绍以下内容：

- 关键字：auto、extern、static、register、const、volatile、restricted、_Thread_local、_Atomic
- 函数：rand()、srand()、time()、malloc()、calloc()、free()
- 如何确定变量的作用域（可见的范围）和生命期（它存在多长时间）
- 设计更复杂的程序

C 语言能让程序员恰到好处地控制程序，这是它的优势之一。程序员通过 C 的内存管理系统指定变量的作用域和生命期，实现对程序的控制。合理使用内存储存数据是设计程序的一个要点。

12.1 存储类别

C 提供了多种不同的模型或存储类别（*storage class*）在内存中储存数据。要理解这些存储类别，先要复习一些概念和术语。

本书目前所有编程示例中使用的数据都储存在内存中。从硬件方面来看，被储存的每个值都占用一定的物理内存，C 语言把这样的一块内存称为对象（*object*）。对象可以储存一个或多个值。一个对象可能并未储存实际的值，但是它在储存适当的值时一定具有相应的大小（面向对象编程中的对象指的是类对象，其定义包括数据和允许对数据进行的操作，C 不是面向对象编程语言）。

从软件方面来看，程序需要一种方法访问对象。这可以通过声明变量来完成：

```
int entity = 3;
```

该声明创建了一个名为 entity 的标识符（*identifier*）。标识符是一个名称，在这种情况下，标识符可以用来指定（*designate*）特定对象的内容。标识符遵循变量的命名规则（第 2 章介绍过）。在该例中，标识符 entity 即是软件（即 C 程序）指定硬件内存中的对象的方式。该声明还提供了储存在对象中的值。

变量名不是指定对象的唯一途径。考虑下面的声明：

```
int * pt = &entity;
int ranks[10];
```

第 1 行声明中，pt 是一个标识符，它指定了一个储存地址的对象。但是，表达式*pt 不是标识符，因为它不是一个名称。然而，它确实指定了一个对象，在这种情况下，它与 entity 指定的对象相同。一般而言，那些指定对象的表达式被称为左值（第 5 章介绍过）。所以，entity 既是标识符也是左值；*pt 既是表达式也是左值。按照这个思路，ranks + 2 * entity 既不是标识符（不是名称），也不是左值（它不指定内存位置上的内容）。但是表达式*(ranks + 2 * entity)是一个左值，因为它的确指定了特定内存位置的值，即 ranks 数组的第 7 个元素。顺带一提，ranks 的声明创建了一个可容纳 10 个 int 类型元素的对象，该数组的每个元素也是一个对象。

所有这些示例中，如果可以使用左值改变对象中的值，该左值就是一个可修改的左值（*modifiable lvalue*）。现在，考虑下面的声明：

```
const char * pc = "Behold a string literal!";
```

程序根据该声明把相应的字符串字面量储存在内存中，内含这些字符值的数组就是一个对象。由于数组中的每个字符都能被单独访问，所以每个字符也是一个对象。该声明还创建了一个标识符为 pc 的对象，储存着字符串的地址。由于可以设置 pc 重新指向其他字符串，所以标识符 pc 是一个可修改的左值。const 只能保证被 pc 指向的字符串内容不被修改，但是无法保证 pc 不指向别的字符串。由于*pc 指定了储存'B'字符的数据对象，所以*pc 是一个左值，但不是一个可修改的左值。与此类似，因为字符串字面量本身指定了储存字符串的对象，所以它也是一个左值，但不是可修改的左值。

可以用存储期（*storage duration*）描述对象，所谓存储期是指对象在内存中保留了多长时间。标识符用于访问对象，可以用作用域（*scope*）和链接（*linkage*）描述标识符，标识符的作用域和链接表明了程序的哪些部分可以使用它。不同的存储类别具有不同的存储期、作用域和链接。标识符可以在源代码的多文件中共享、可用于特定文件的任意函数中、可仅限于特定函数中使用，甚至只在函数中的某部分使用。对象可存在于程序的执行期，也可以仅存在于它所在函数的执行期。对于并发编程，对象可以在特定线程的执行期存在。可以通过函数调用的方式显式分配和释放内存。

我们先学习作用域、链接和存储期的含义，再介绍具体的存储类别。

12.1.1　作用域

作用域描述程序中可访问标识符的区域。一个 C 变量的作用域可以是块作用域、函数作用域、函数原型作用域或文件作用域。到目前为止，本书程序示例中使用的变量几乎都具有块作用域。块是用一对花括号括起来的代码区域。例如，整个函数体是一个块，函数中的任意复合语句也是一个块。定义在块中的变量具有块作用域（*block scope*），块作用域变量的可见范围是从定义处到包含该定义的块的末尾。另外，虽然函数的形式参数声明在函数的左花括号之前，但是它们也具有块作用域，属于函数体这个块。所以到目前为止，我们使用的局部变量（包括函数的形式参数）都具有块作用域。因此，下面代码中的变量 cleo 和 patrick 都具有块作用域：

```
double blocky(double cleo)
{
    double patrick = 0.0;
    ...
    return patrick;
}
```

声明在内层块中的变量，其作用域仅局限于该声明所在的块：

```
double blocky(double cleo)
{
    double patrick = 0.0;
    int i;
    for (i = 0; i < 10; i++)
    {
        double q = cleo * i; // q的作用域开始
        ...
        patrick *= q;
    }                        // q的作用域结束
    ...
    return patrick;
}
```

在该例中，q 的作用域仅限于内层块，只有内层块中的代码才能访问 q。

以前，具有块作用域的变量都必须声明在块的开头。C99 标准放宽了这一限制，允许在块中的任意位置声明变量。因此，对于 for 的循环头，现在可以这样写：

```
for (int i = 0; i < 10; i++)
    printf("A C99 feature: i = %d", i);
```

为适应这个新特性，C99 把块的概念扩展到包括 for 循环、while 循环、do while 循环和 if 语句所控制的代码，即使这些代码没有用花括号括起来，也算是块的一部分。所以，上面 for 循环中的变量 i 被视为 for 循环块的一部分，它的作用域仅限于 for 循环。一旦程序离开 for 循环，就不能再访问 i。

函数作用域（*function scope*）仅用于 goto 语句的标签。这意味着即使一个标签首次出现在函数的内层块中，它的作用域也延伸至整个函数。如果在两个块中使用相同的标签会很混乱，标签的函数作用域防止了这样的事情发生。

函数原型作用域（*function prototype scope*）用于函数原型中的形参名（变量名），如下所示：

```
int mighty(int mouse, double large);
```

函数原型作用域的范围是从形参定义处到原型声明结束。这意味着，编译器在处理函数原型中的形参时只关心它的类型，而形参名（如果有的话）通常无关紧要。而且，即使有形参名，也不必与函数定义中的形参名相匹配。只有在变长数组中，形参名才有用：

```
void use_a_VLA(int n, int m, ar[n][m]);
```

方括号中必须使用在函数原型中已声明的名称。

变量的定义在函数的外面，具有文件作用域（*file scope*）。具有文件作用域的变量，从它的定义处到该定义所在文件的末尾均可见。考虑下面的例子：

```
#include <stdio.h>
int units = 0;         /* 该变量具有文件作用域 */
void critic(void);
int main(void)
{
    ...
}
void critic(void)
{
    ...
}
```

这里，变量 units 具有文件作用域，main()和 critic()函数都可以使用它（更准确地说，units 具有外部链接文件作用域，稍后讲解）。由于这样的变量可用于多个函数，所以文件作用域变量也称为全局变量（*global variable*）。

注意　翻译单元和文件

你认为的多个文件在编译器中可能以一个文件出现。例如，通常在源代码（.c 扩展名）中包含一个或多个头文件（.h 扩展名）。头文件会依次包含其他头文件，所以会包含多个单独的物理文件。但是，C 预处理实际上是用包含的头文件内容替换#include 指令。所以，编译器源代码文件和所有的头文件都看成是一个包含信息的单独文件。这个文件被称为翻译单元（*translation unit*）。描述一个具有文件作用域的变量时，它的实际可见范围是整个翻译单元。如果程序由多个源代码文件组成，那么该程序也将由多个翻译单元组成。每个翻译单元均对应一个源代码文件和它所包含的文件。

12.1.2　链接

接下来，我们介绍链接。C 变量有 3 种链接属性：外部链接、内部链接或无链接。具有块作用域、函数作用域或函数原型作用域的变量都是无链接变量。这意味着这些变量属于定义它们的块、函数或原型私有。具有文件作用域的变量可以是外部链接或内部链接。外部链接变量可以在多文件程序中使用，内部链接变量只能在一个翻译单元中使用。

> **注意　正式和非正式术语**
>
> C 标准用“内部链接的文件作用域”描述仅限于一个翻译单元（即一个源代码文件和它所包含的头文件）的作用域，用“外部链接的文件作用域”描述可延伸至其他翻译单元的作用域。但是，对程序员而言这些术语太长了。一些程序员把“内部链接的文件作用域”简称为“文件作用域”，把“外部链接的文件作用域”简称为“全局作用域”或“程序作用域”。

如何知道文件作用域变量是内部链接还是外部链接？可以查看外部定义中是否使用了存储类别说明符 static：

```
int giants = 5;            // 文件作用域，外部链接
static int dodgers = 3;    // 文件作用域，内部链接
int main()
{
    ...
}
...
```

该文件和同一程序的其他文件都可以使用变量 giants。而变量 dodgers 属文件私有，该文件中的任意函数都可使用它。

12.1.3　存储期

作用域和链接描述了标识符的可见性。存储期描述了通过这些标识符访问的对象的生存期。C 对象有 4 种存储期：静态存储期、线程存储期、自动存储期、动态分配存储期。

如果对象具有静态存储期，那么它在程序的执行期间一直存在。文件作用域变量具有静态存储期。注意，对于文件作用域变量，关键字 static 表明了其链接属性，而非存储期。以 static 声明的文件作用域变量具有内部链接。但是无论是内部链接还是外部链接，所有的文件作用域变量都具有静态存储期。

线程存储期用于并发程序设计，程序执行可被分为多个线程。具有线程存储期的对象，从被声明时到线程结束一直存在。以关键字_Thread_local 声明一个对象时，每个线程都获得该变量的私有备份。

块作用域的变量通常都具有自动存储期。当程序进入定义这些变量的块时，为这些变量分配内存；当退出这个块时，释放刚才为变量分配的内存。这种做法相当于把自动变量占用的内存视为一个可重复使用的工作区或暂存区。例如，一个函数调用结束后，其变量占用的内存可用于储存下一个被调用函数的变量。

变长数组稍有不同，它们的存储期从声明处到块的末尾，而不是从块的开始处到块的末尾。

我们到目前为止使用的局部变量都是自动类别。例如，在下面的代码中，变量 number 和 index 在每次调用 bore()函数时被创建，在离开函数时被销毁：

```
void bore(int number)
{
    int index;
    for (index = 0; index < number; index++)
        puts("They don't make them the way they used to.\n");
    return 0;
}
```

然而，块作用域变量也能具有静态存储期。为了创建这样的变量，要把变量声明在块中，且在声明前面加上关键字 static：

```
void more(int number)
{
    int index;
    static int ct = 0;
    ...
    return 0;
}
```

这里，变量 ct 储存在静态内存中，它从程序被载入到程序结束期间都存在。但是，它的作用域定义在 more()函数块中。只有在执行该函数时，程序才能使用 ct 访问它所指定的对象（但是，该函数可以给其他函数提供该存储区的地址以便间接访问该对象，例如通过指针形参或返回值）。

C 使用作用域、链接和存储期为变量定义了多种存储方案。本书不涉及并发程序设计，所以不再赘述这方面的内容。已分配存储期在本章后面介绍。因此，剩下 5 种存储类别：自动、寄存器、静态块作用域、静态外部链接、静态内部链接，如表 12.1 所列。现在，我们已经介绍了作用域、链接和存储期，接下来将详细讨论这些存储类别。

表 12.1　5 种存储类别

存储类别	存储期	作用域	链接	声明方式
自动	自动	块	无	块内
寄存器	自动	块	无	块内，使用关键字 register
静态外部链接	静态	文件	外部	所有函数外
静态内部链接	静态	文件	内部	所有函数外，使用关键字 static
静态无链接	静态	块	无	块内，使用关键字 static

12.1.4　自动变量

属于自动存储类别的变量具有自动存储期、块作用域且无链接。默认情况下，声明在块或函数头中的任何变量都属于自动存储类别。为了更清楚地表达你的意图（例如，为了表明有意覆盖一个外部变量定义，或者强调不要把该变量改为其他存储类别），可以显式使用关键字 auto，如下所示：

```
int main(void)
{
    auto int plox;
```

关键字 auto 是存储类别说明符（*storage-class specifier*）。auto 关键字在 C++中的用法完全不同，如果编写 C/C++兼容的程序，最好不要使用 auto 作为存储类别说明符。

块作用域和无链接意味着只有在变量定义所在的块中才能通过变量名访问该变量（当然，参数用于传递变量的值和地址给另一个函数，但是这是间接的方法）。另一个函数可以使用同名变量，但是该变量是储存在不同内存位置上的另一个变量。

变量具有自动存储期意味着，程序在进入该变量声明所在的块时变量存在，程序在退出该块时变量消失。原来该变量占用的内存位置现在可做他用。

接下来分析一下嵌套块的情况。块中声明的变量仅限于该块及其包含的块使用。

```
int loop(int n)
{
    int m; // m 的作用域
    scanf("%d", &m);
    {
        int i; // m 和 i 的作用域
        for (i = m; i < n; i++)
            puts("i is local to a sub-block\n");
    }
    return m; // m 的作用域，i 已经消失
}
```

在上面的代码中，i 仅在内层块中可见。如果在内层块的前面或后面使用 i，编译器会报错。通常，在设计程序时用不到这个特性。然而，如果这个变量仅供该块使用，那么在块中就近定义该变量也很方便。这样，可以在靠近使用变量的地方记录其含义。另外，这样的变量只有在使用时才占用内存。变量 n 和 m 分别定义在函数头和外层块中，它们的作用域是整个函数，而且在调用函数到函数结束期间都一直存在。

如果内层块中声明的变量与外层块中的变量同名会怎样？内层块会隐藏外层块的定义。但是离开内层块后，外层块变量的作用域又回到了原来的作用域。程序清单 12.1 演示了这一过程。

程序清单 12.1 hiding.c **程序**

```
// hiding.c -- 块中的变量
#include <stdio.h>
int main()
{
    int x = 30;            // 原始的 x

    printf("x in outer block: %d at %p\n", x, &x);
    {
        int x = 77;        // 新的 x，隐藏了原始的 x
        printf("x in inner block: %d at %p\n", x, &x);
    }
    printf("x in outer block: %d at %p\n", x, &x);
    while (x++ < 33)       // 原始的 x
    {
        int x = 100;       // 新的 x，隐藏了原始的 x
        x++;
        printf("x in while loop: %d at %p\n", x, &x);
    }
    printf("x in outer block: %d at %p\n", x, &x);

    return 0;
}
```

下面是该程序的输出：

```
x in outer block: 30 at 0x7fff5fbff8c8
x in inner block: 77 at 0x7fff5fbff8c4
```

```
x in outer block: 30 at 0x7fff5fbff8c8
x in while loop: 101 at 0x7fff5fbff8c0
x in while loop: 101 at 0x7fff5fbff8c0
x in while loop: 101 at 0x7fff5fbff8c0
x in outer block: 34 at 0x7fff5fbff8c8
```

首先，程序创建了变量 `x` 并初始化为 `30`，如第 1 条 `printf()`语句所示。然后，定义了一个新的变量 `x`，并设置为 `77`，如第 2 条 `printf()`语句所示。根据显示的地址可知，新变量隐藏了原始的 `x`。第 3 条 `printf()`语句位于第 1 个内层块后面，显示的是原始的 `x` 的值，这说明原始的 `x` 既没有消失也不曾改变。

也许该程序最难懂的是 `while` 循环。`while` 循环的测试条件中使用的是原始的 `x`：

```
while(x++ < 33)
```

在该循环中，程序创建了第 3 个 `x` 变量，该变量只定义在 `while` 循环中。所以，当执行到循环体中的 `x++`时，递增为 `101` 的是新的 `x`，然后 `printf()`语句显示了该值。每轮迭代结束，新的 `x` 变量就消失。然后循环的测试条件使用并递增原始的 `x`，再次进入循环体，再次创建新的 `x`。在该例中，这个 `x` 被创建和销毁了 3 次。注意，该循环必须在测试条件中递增 `x`，因为如果在循环体中递增 `x`，那么递增的是循环体中创建的 `x`，而非测试条件中使用的原始 `x`。

我们使用的编译器在创建 `while` 循环体中的 `x` 时，并未复用内层块中 `x` 占用的内存，但是有些编译器会这样做。

该程序示例的用意不是鼓励读者要编写类似的代码（根据 C 的命名规则，要想出别的变量名并不难），而是为了解释在内层块中定义变量的具体情况。

1. 没有花括号的块

前面提到一个 C99 特性：作为循环或 `if` 语句的一部分，即使不使用花括号（`{}`），也是一个块。更完整地说，整个循环是它所在块的子块（*sub-block*），循环体是整个循环块的子块。与此类似，`if` 语句是一个块，与其相关联的子语句是 `if` 语句的子块。这些规则会影响到声明的变量和这些变量的作用域。程序清单 12.2 演示了 `for` 循环中该特性的用法。

程序清单 12.2　`forc99.c` 程序

```
// forc99.c -- 新的 C99 块规则
#include <stdio.h>
int main()
{
    int n = 8;

    printf("   Initially, n = %d at %p\n", n, &n);
    for (int n = 1; n < 3; n++)
        printf("      loop 1: n = %d at %p\n", n, &n);
    printf("After loop 1, n = %d at %p\n", n, &n);
    for (int n = 1; n < 3; n++)
    {
        printf(" loop 2 index n = %d at %p\n", n, &n);
        int n = 6;
        printf("      loop 2: n = %d at %p\n", n, &n);
        n++;
    }
    printf("After loop 2, n = %d at %p\n", n, &n);

    return 0;
}
```

假设编译器支持 C 语言的这个新特性，该程序的输出如下：

```
    Initially, n = 8 at 0x7fff5fbff8c8
        loop 1: n = 1 at 0x7fff5fbff8c4
        loop 1: n = 2 at 0x7fff5fbff8c4
 After loop 1, n = 8 at 0x7fff5fbff8c8
  loop 2 index n = 1 at 0x7fff5fbff8c0
        loop 2: n = 6 at 0x7fff5fbff8bc
  loop 2 index n = 2 at 0x7fff5fbff8c0
        loop 2: n = 6 at 0x7fff5fbff8bc
 After loop 2, n = 8 at 0x7fff5fbff8c8
```

第 1 个 `for` 循环头中声明的 `n`，其作用域作用至循环末尾，而且隐藏了原始的 `n`。但是，离开循环后，原始的 `n` 又起作用了。

第 2 个 `for` 循环头中声明的 `n` 作为循环的索引，隐藏了原始的 `n`。然后，在循环体中又声明了一个 `n`，隐藏了索引 `n`。结束一轮迭代后，声明在循环体中的 `n` 消失，循环头使用索引 `n` 进行测试。当整个循环结束时，原始的 `n` 又起作用了。再次提醒读者注意，没必要在程序中使用相同的变量名。如果用了，各变量的情况如上所述。

注意　支持 C99 和 C11

有些编译器并不支持 C99/C11 的这些作用域规则（Microsoft Visual Studio 2012 就是其中之一）。有些编译会提供激活这些规则的选项。例如，撰写本书时，gcc 默认支持了 C99 的许多特性，但是要用`-std=c99`选项激活程序清单 12.2 中使用的特性:

```
gcc -std=c99 forc99.c
```

与此类似，gcc 或 clang 都要使用`-std=c1x`或`-std=c11`选项，才支持 C11 特性。

2. 自动变量的初始化

自动变量不会初始化，除非显式初始化它。考虑下面的声明：

```
int main(void)
{
     int repid;
     int tents = 5;
```

`tents` 变量被初始化为 5，但是 `repid` 变量的值是之前占用分配给 `repid` 的空间中的任意值（如果有的话），别指望这个值是 0。可以用非常量表达式（*non-constant expression*）初始化自动变量，前提是所用的变量已在前面定义过：

```
int main(void)
{
     int ruth = 1;
     int rance = 5 * ruth;  // 使用之前定义的变量
```

12.1.5 寄存器变量

变量通常储存在计算机内存中。如果幸运的话，寄存器变量储存在 CPU 的寄存器中，或者概括地说，储存在最快的可用内存中。与普通变量相比，访问和处理这些变量的速度更快。由于寄存器变量储存在寄存器而非内存中，所以无法获取寄存器变量的地址。绝大多数方面，寄存器变量和自动变量都一样。也就是说，它们都是块作用域、无链接和自动存储期。使用存储类别说明符 `register` 便可声明寄存器变量：

```
int main(void)
{
    register int quick;
```

我们刚才说“如果幸运的话”，是因为声明变量为 register 类别与直接命令相比更像是一种请求。编译器必须根据寄存器或最快可用内存的数量衡量你的请求，或者直接忽略你的请求，所以可能不会如你所愿。在这种情况下，寄存器变量就变成普通的自动变量。即使是这样，仍然不能对该变量使用地址运算符。

在函数头中使用关键字 register，便可请求形参是寄存器变量：

```
void macho(register int n)
```

可声明为 register 的数据类型有限。例如，处理器中的寄存器可能没有足够大的空间来储存 double 类型的值。

12.1.6　块作用域的静态变量

静态变量（*static variable*）听起来自相矛盾，像是一个不可变的变量。实际上，静态的意思是该变量在内存中原地不动，并不是说它的值不变。具有文件作用域的变量自动具有（也必须是）静态存储期。前面提到过，可以创建具有静态存储期、块作用域的局部变量。这些变量和自动变量一样，具有相同的作用域，但是程序离开它们所在的函数后，这些变量不会消失。也就是说，这种变量具有块作用域、无链接，但是具有静态存储期。计算机在多次函数调用之间会记录它们的值。在块中（提供块作用域和无链接）以存储类别说明符 static（提供静态存储期）声明这种变量。程序清单 12.3 演示了一个这样的例子。

程序清单 12.3　loc_stat.c 程序

```
/* loc_stat.c -- 使用局部静态变量 */
#include <stdio.h>
void trystat(void);

int main(void)
{
    int count;

    for (count = 1; count <= 3; count++)
    {
        printf("Here comes iteration %d:\n", count);
        trystat();
    }

    return 0;
}

void trystat(void)
{
    int fade = 1;
    static int stay = 1;

    printf("fade = %d and stay = %d\n", fade++, stay++);
}
```

注意，trystat()函数先打印再递增变量的值。该程序的输出如下：

```
Here comes iteration 1:
fade = 1 and stay = 1
```

```
Here comes iteration 2:
fade = 1 and stay = 2
Here comes iteration 3:
fade = 1 and stay = 3
```

静态变量 stay 保存了它被递增 1 后的值，但是 fade 变量每次都是 1。这表明了初始化的不同：每次调用 trystat()都会初始化 fade，但是 stay 只在编译 strstat()时被初始化一次。如果未显式初始化静态变量，它们会被初始化为 0。

下面两个声明很相似：

```
int fade = 1;
static int stay = 1;
```

第 1 条声明确实是 trystat()函数的一部分，每次调用该函数时都会执行这条声明。这是运行时行为。第 2 条声明实际上并不是 trystat()函数的一部分。如果逐步调试该程序会发现，程序似乎跳过了这条声明。这是因为静态变量和外部变量在程序被载入内存时已执行完毕。把这条声明放在 trystat()函数中是为了告诉编译器只有 trystat()函数才能看到该变量。这条声明并未在运行时执行。

不能在函数的形参中使用 static：

```
int wontwork(static int flu); // 不允许
```

“局部静态变量”是描述具有块作用域的静态变量的另一个术语。阅读一些老的 C 文献时会发现，这种存储类别被称为内部静态存储类别（*internal static storage class*）。这里的内部指的是函数内部，而非内部链接。

12.1.7　外部链接的静态变量

外部链接的静态变量具有文件作用域、外部链接和静态存储期。该类别有时称为外部存储类别（*external storage class*），属于该类别的变量称为外部变量（*external variable*）。把变量的定义性声明（*defining declaration*）放在在所有函数的外面便创建了外部变量。当然，为了指出该函数使用了外部变量，可以在函数中用关键字 extern 再次声明。如果一个源代码文件使用的外部变量定义在另一个源代码文件中，则必须用 extern 在该文件中声明该变量。如下所示：

```
int Errupt;                 /* 外部定义的变量 */
double Up[100];             /* 外部定义的数组 */
extern char Coal;           /* 如果 Coal 被定义在另一个文件， */
                            /*则必须这样声明*/
void next(void);
int main(void)
{
    extern int Errupt;      /* 可选的声明*/

    extern double Up[];     /* 可选的声明*/
    ...
}
void next(void)
{
    ...
}
```

注意，在 main()中声明 Up 数组时（这是可选的声明）不用指明数组大小，因为第 1 次声明已经提供了数组大小信息。main()中的两条 extern 声明完全可以省略，因为外部变量具有文件作用域，所以 Errupt 和 Up 从声明处到文件结尾都可见。它们出现在那里，仅为了说明 main()函数要使用这两个变量。

如果省略掉函数中的 extern 关键字，相当于创建了一个自动变量。去掉下面声明中的 extern：

```
extern int Errupt;
```

便成为：

```
int Errupt;
```

这使得编译器在 main()中创建了一个名为 Errupt 的自动变量。它是一个独立的局部变量，与原来的外部变量 Errupt 不同。该局部变量仅 main()中可见，但是外部变量 Errupt 对于该文件的其他函数（如 next()）也可见。简而言之，在执行块中的语句时，块作用域中的变量将“隐藏”文件作用域中的同名变量。如果不得已要使用与外部变量同名的局部变量，可以在局部变量的声明中使用 auto 存储类别说明符明确表达这种意图。

外部变量具有静态存储期。因此，无论程序执行到 main()、next()还是其他函数，数组 Up 及其值都一直存在。

下面 3 个示例演示了外部和自动变量的一些使用情况。示例 1 中有一个外部变量 Hocus。该变量对 main()和 magic()均可见。

```
/* 示例 1 */
int Hocus;
int magic();
int main(void)
{
    extern int Hocus;  // Hocus 之前已声明为外部变量
    ...
}
int magic()
{
    extern int Hocus;  // 与上面的 Hocus 是同一个变量
    ...
}
```

示例 2 中有一个外部变量 Hocus，对两个函数均可见。这次，在默认情况下对 magic()可见。

```
/*示例 2  */
int Hocus;
int magic();
int main(void)
{
    extern int Hocus; // Hocus 之前已声明为外部变量
    ...
}
int magic()
{
                          //并未在该函数中声明 Hocus，但是仍可使用该变量
    ...
}
```

在示例 3 中，创建了 4 个独立的变量。main()中的 Hocus 变量默认是自动变量，属于 main()私有。magic()中的 Hocus 变量被显式声明为自动，只有 magic()可用。外部变量 Houcus 对 main()和 magic()均不可见，但是对该文件中未创建局部 Hocus 变量的其他函数可见。最后，Pocus 是外部变量，magic()可见，但是 main()不可见，因为 Pocus 被声明在 main()后面。

```
/* 示例 3 */
int Hocus;
int magic();
```

```
int main(void)
{
  int Hocus; // 声明Hocus，默认是自动变量
     ...
}
int Pocus;
int magic()
{
     auto int Hocus; //把局部变量Hocus显式声明为自动变量
     ...
}
```

这 3 个示例演示了外部变量的作用域是：从声明处到文件结尾。除此之外，还说明了外部变量的生命期。外部变量 Hocus 和 Pocus 在程序运行中一直存在，因为它们不受限于任何函数，不会在某个函数返回后就消失。

1. 初始化外部变量

外部变量和自动变量类似，也可以被显式初始化。与自动变量不同的是，如果未初始化外部变量，它们会被自动初始化为 0。这一原则也适用于外部定义的数组元素。与自动变量的情况不同，只能使用常量表达式初始化文件作用域变量：

```
int x = 10;                  // 没问题，10是常量
int y = 3 + 20;              // 没问题，用于初始化的是常量表达式
size_t z = sizeof(int);      //没问题，用于初始化的是常量表达式
int x2 = 2 * x;              // 不行，x是变量
```

（只要不是变长数组，sizeof 表达式可被视为常量表达式。）

2. 使用外部变量

下面来看一个使用外部变量的示例。假设有两个函数main()和critic()，它们都要访问变量units。可以把 units 声明在这两个函数的上面，如程序清单 12.4 所示（注意：该例的目的是演示外部变量的工作原理，并非它的典型用法）。

程序清单 12.4　global.c 程序

```
/* global.c  -- 使用外部变量 */
#include <stdio.h>
int units = 0;        /* 外部变量  */
void critic(void);
int main(void)
{
     extern int units;  /* 可选的重复声明 */

     printf("How many pounds to a firkin of butter?\n");
     scanf("%d", &units);
     while (units != 56)
          critic();
     printf("You must have looked it up!\n");

     return 0;
}

void critic(void)
{
```

```
    /* 删除了可选的重复声明 */
    printf("No luck, my friend. Try again.\n");
    scanf("%d", &units);
}
```

下面是该程序的输出示例：

```
How many pounds to a firkin of butter?
14
No luck, my friend. Try again.
56
You must have looked it up!
```

注意，critic()是如何读取 units 的第 2 个值的。当 while 循环结束时，main()也知道 units 的新值。所以 main()函数和 critic()都可以通过标识符 units 访问相同的变量。用 C 的术语来描述是，units 具有文件作用域、外部链接和静态存储期。

把 units 定义在所有函数定义外面（即外部），units 便是一个外部变量，对 units 定义下面的所有函数均可见。因此，critics()可以直接使用 units 变量。

类似地，main()也可直接访问 units。但是，main()中确实有如下声明：

```
extern int units;
```

本例中，以上声明主要是为了指出该函数要使用这个外部变量。存储类别说明符 extern 告诉编译器，该函数中任何使用 units 的地方都引用同一个定义在函数外部的变量。再次强调，main()和 critic()使用的都是外部定义的 units。

3. 外部名称

C99 和 C11 标准都要求编译器识别局部标识符的前 63 个字符和外部标识符的前 31 个字符。这修订了以前的标准，即编译器识别局部标识符前 31 个字符和外部标识符前 6 个字符。你所用的编译器可能还执行以前的规则。外部变量名比局部变量名的规则严格，是因为外部变量名还要遵循局部环境规则，所受的限制更多。

4. 定义和声明

下面进一步介绍定义变量和声明变量的区别。考虑下面的例子：

```
int tern = 1; /* tern 被定义 */
main()
{
    extern int tern; /* 使用在别处定义的 tern */
```

这里，tern 被声明了两次。第 1 次声明为变量预留了存储空间，该声明构成了变量的定义。第 2 次声明只告诉编译器使用之前已创建的 tern 变量，所以这不是定义。第 1 次声明被称为*定义式声明*（*defining declaration*），第 2 次声明被称为*引用式声明*（*referencing declaration*）。关键字 extern 表明该声明不是定义，因为它指示编译器去别处查询其定义。

假设这样写：

```
extern int tern;
int main(void)
{
```

编译器会假设 tern 实际的定义在该程序的别处，也许在别的文件中。该声明并不会引起分配存储空间。因此，不要用关键字 extern 创建外部定义，只用它来引用现有的外部定义。

外部变量只能初始化一次，且必须在定义该变量时进行。假设有下面的代码：

```
// file_one.c
char permis = 'N';
...
// file_two.c
extern char permis = 'Y'; /* 错误 */
```

file_two 中的声明是错误的，因为 file_one.c 中的定义式声明已经创建并初始化了 permis。

12.1.8　内部链接的静态变量

该存储类别的变量具有静态存储期、文件作用域和内部链接。在所有函数外部（这点与外部变量相同），用存储类别说明符 static 定义的变量具有这种存储类别：

```
static int svil = 1;   // 静态变量，内部链接
int main(void)
{
```

这种变量过去称为外部静态变量（*external static variable*），但是这个术语有点自相矛盾（这些变量具有内部链接）。但是，没有合适的新简称，所以只能用内部链接的静态变量（*static variable with internal linkage*）。普通的外部变量可用于同一程序中任意文件中的函数，但是内部链接的静态变量只能用于同一个文件中的函数。可以使用存储类别说明符 extern，在函数中重复声明任何具有文件作用域的变量。这样的声明并不会改变其链接属性。考虑下面的代码：

```
int traveler = 1;          // 外部链接
static int stayhome = 1;   // 内部链接
int main()
{
    extern int traveler;   // 使用定义在别处的 traveler
    extern int stayhome;   // 使用定义在别处的 stayhome
    ...
```

对于该程序所在的翻译单元，trveler 和 stayhome 都具有文件作用域，但是只有 traveler 可用于其他翻译单元（因为它具有外部链接）。这两个声明都使用了 extern 关键字，指明了 main()中使用的这两个变量的定义都在别处，但是这并未改变 stayhome 的内部链接属性。

12.1.9　多文件

只有当程序由多个翻译单元组成时，才体现区别内部链接和外部链接的重要性。接下来简要介绍一下。

复杂的 C 程序通常由多个单独的源代码文件组成。有时，这些文件可能要共享一个外部变量。C 通过在一个文件中进行定义式声明，然后在其他文件中进行引用式声明来实现共享。也就是说，除了一个定义式声明外，其他声明都要使用 extern 关键字。而且，只有定义式声明才能初始化变量。

注意，如果外部变量定义在一个文件中，那么其他文件在使用该变量之前必须先声明它（用 extern 关键字）。也就是说，在某文件中对外部变量进行定义式声明只是单方面允许其他文件使用该变量，其他文件在用 extern 声明之前不能直接使用它。

过去，不同的编译器遵循不同的规则。例如，许多 UNIX 系统允许在多个文件中不使用 extern 关键字声明变量，前提是只有一个带初始化的声明。编译器会把文件中一个带初始化的声明视为该变量的定义。

12.1.10 存储类别说明符

读者可能已经注意到了，关键字 static 和 extern 的含义取决于上下文。C 语言有 6 个关键字作为存储类别说明符：auto、register、static、extern、_Thread_local 和 typedef。typedef 关键字与任何内存存储无关，把它归于此类有一些语法上的原因。尤其是，在绝大多数情况下，不能在声明中使用多个存储类别说明符，所以这意味着不能使用多个存储类别说明符作为 typedef 的一部分。唯一例外的是_Thread_local，它可以和 static 或 extern 一起使用。

auto 说明符表明变量是自动存储期，只能用于块作用域的变量声明中。由于在块中声明的变量本身就具有自动存储期，所以使用 auto 主要是为了明确表达要使用与外部变量同名的局部变量的意图。

register 说明符也只用于块作用域的变量，它把变量归为寄存器存储类别，请求最快速度访问该变量。同时，还保护了该变量的地址不被获取。

用 static 说明符创建的对象具有静态存储期，载入程序时创建对象，当程序结束时对象消失。如果 static 用于文件作用域声明，作用域受限于该文件。如果 static 用于块作用域声明，作用域则受限于该块。因此，只要程序在运行对象就存在并保留其值，但是只有在执行块内的代码时，才能通过标识符访问。块作用域的静态变量无链接。文件作用域的静态变量具有内部链接。

extern 说明符表明声明的变量定义在别处。如果包含 extern 的声明具有文件作用域，则引用的变量必须具有外部链接。如果包含 extern 的声明具有块作用域，则引用的变量可能具有外部链接或内部链接，这接取决于该变量的定义式声明。

> **小结：存储类别**
>
> 自动变量具有块作用域、无链接、自动存储期。它们是局部变量，属于其定义所在块（通常指函数）私有。寄存器变量的属性和自动变量相同，但是编译器会使用更快的内存或寄存器储存它们。不能获取寄存器变量的地址。
>
> 具有静态存储期的变量可以具有外部链接、内部链接或无链接。在同一个文件所有函数的外部声明的变量是外部变量，具有文件作用域、外部链接和静态存储期。如果在这种声明前面加上关键字 static，那么其声明的变量具有文件作用域、内部链接和静态存储期。如果在函数中用 static 声明一个变量，则该变量具有块作用域、无链接、静态存储期。
>
> 具有自动存储期的变量，程序在进入该变量的声明所在块时才为其分配内存，在退出该块时释放之前分配的内存。如果未初始化，自动变量中是垃圾值。程序在编译时为具有静态存储期的变量分配内存，并在程序的运行过程中一直保留这块内存。如果未初始化，这样的变量会被设置为 0。
>
> 具有块作用域的变量是局部的，属于包含该声明的块私有。具有文件作用域的变量对文件（或翻译单元）中位于其声明后面的所有函数可见。具有外部链接的文件作用域变量，可用于该程序的其他翻译单元。具有内部链接的文件作用域变量，只能用于其声明所在的文件内。

下面用一个简短的程序使用了 5 种存储类别。该程序包含两个文件（程序清单 12.5 和程序清单 12.6），所以必须使用多文件编译（参见第 9 章或参看编译器的指导手册）。该示例仅为了让读者熟悉 5 种存储类别的用法，并不是提供设计模型，好的设计可以不需要使用文件作用域变量。

程序清单 12.5 parta.c 程序

```
// parta.c --- 不同的存储类别
```

```
// 与 partb.c 一起编译
#include <stdio.h>
void report_count();
void accumulate(int k);
int count = 0;          // 文件作用域，外部链接

int main(void)
{
    int value;          // 自动变量
    register int i;     // 寄存器变量

    printf("Enter a positive integer (0 to quit): ");
    while (scanf("%d", &value) == 1 && value > 0)
    {
        ++count;        // 使用文件作用域变量
        for (i = value; i >= 0; i--)
            accumulate(i);
        printf("Enter a positive integer (0 to quit): ");
    }
    report_count();

    return 0;
}

void report_count()
{
    printf("Loop executed %d times\n", count);
}
```

程序清单 12.6　partb.c 程序

```
// partb.c -- 程序的其余部分
// 与 parta.c 一起编译
#include <stdio.h>

extern int count;           // 引用式声明，外部链接

static int total = 0;       // 静态定义，内部链接
void accumulate(int k);     // 函数原型

void accumulate(int k) // k 具有块作用域，无链接
{
    static int subtotal = 0;    // 静态，无链接

    if (k <= 0)
    {
        printf("loop cycle: %d\n", count);
        printf("subtotal: %d; total: %d\n", subtotal, total);
        subtotal = 0;
    }
    else
    {
        subtotal += k;
```

```
            total += k;
        }
    }
```

在该程序中，块作用域的静态变量 subtotal 统计每次 while 循环传入 accumulate()函数的总数，具有文件作用域、内部链接的变量 total 统计所有传入 accumulate()函数的总数。当传入负值时，accumulate()函数报告 total 和 subtotal 的值，并在报告后重置 subtotal 为 0。由于 parta.c 调用了 accumulate()函数，所以必须包含 accumulate()函数的原型。而 partb.c 只包含了 accumulate()函数的定义，并未在文件中调用该函数，所以其原型为可选（即省略原型也不影响使用）。该函数使用了外部变量 count 统计 main()中的 while 循环迭代的次数（顺带一提，对于该程序，没必要使用外部变量把 parta.c 和 partb.c 的代码弄得这么复杂）。在 parta.c 中，main()和 report_count()共享 count。

下面是程序的运行示例：

```
Enter a positive integer (0 to quit): 5
loop cycle: 1
subtotal: 15; total: 15
Enter a positive integer (0 to quit): 10
loop cycle: 2
subtotal: 55; total: 70
Enter a positive integer (0 to quit): 2
loop cycle: 3
subtotal: 3; total: 73
Enter a positive integer (0 to quit): 0
Loop executed 3 times
```

12.1.11 存储类别和函数

函数也有存储类别，可以是外部函数（默认）或静态函数。C99 新增了第 3 种类别——内联函数，将在第 16 章中介绍。外部函数可以被其他文件的函数访问，但是静态函数只能用于其定义所在的文件。假设一个文件中包含了以下函数原型：

```
double gamma(double);      /* 该函数默认为外部函数 */
static double beta(int, int);
extern double delta(double, int);
```

在同一个程序中，其他文件中的函数可以调用 gamma()和 delta()，但是不能调用 beta()，因为以 static 存储类别说明符创建的函数属于特定模块私有。这样做避免了名称冲突的问题，由于 beta()受限于它所在的文件，所以在其他文件中可以使用与之同名的函数。

通常的做法是：用 extern 关键字声明定义在其他文件中的函数。这样做是为了表明当前文件中使用的函数被定义在别处。除非使用 static 关键字，否则一般函数声明都默认为 extern。

12.1.12 存储类别的选择

对于“使用哪种存储类别”的回答绝大多数是“自动存储类别”，要知道默认类别就是自动存储类别。初学者会认为外部存储类别很不错，为何不把所有的变量都设置成外部变量，这样就不必使用参数和指针在函数间传递信息了。然而，这背后隐藏着一个陷阱。如果这样做，A()函数可能违背你的意图，私下修改 B()函数使用的变量。多年来，无数程序员的经验表明，随意使用外部存储类别的变量导致的后果远远超过了它所带来的便利。

唯一例外的是 const 数据。因为它们在初始化后就不会被修改，所以不用担心它们被意外篡改：

```
const int DAYS = 7;
const char * MSGS[3] = {"Yes", "No", Maybe"};
```

保护性程序设计的黄金法则是："按需知道"原则。尽量在函数内部解决该函数的任务，只共享那些需要共享的变量。除自动存储类别外，其他存储类别也很有用。不过，在使用某类别之前先要考虑一下是否有必要这样做。

12.2　随机数函数和静态变量

学习了不同存储类别的概念后，我们来看几个相关的程序。首先，来看一个使用内部链接的静态变量的函数：随机数函数。ANSI C 库提供了 rand() 函数生成随机数。生成随机数有多种算法，ANSI C 允许 C 实现针对特定机器使用最佳算法。然而，ANSI C 标准还提供了一个可移植的标准算法，在不同系统中生成相同的随机数。实际上，rand() 是"伪随机数生成器"，意思是可预测生成数字的实际序列。但是，数字在其取值范围内均匀分布。

为了看清楚程序内部的情况，我们使用可移植的 ANSI 版本，而不是编译器内置的 rand() 函数。可移植版本的方案开始于一个"种子"数字。该函数使用该种子生成新的数，这个新数又成为新的种子。然后，新种子可用于生成更新的种子，以此类推。该方案要行之有效，随机数函数必须记录它上一次被调用时所使用的种子。这里需要一个静态变量。程序清单 12.7 演示了版本 0（稍后给出版本 1）。

程序清单 12.7　rand0.c 函数文件

```
/* rand0.c --生成随机数*/
/* 使用 ANSI C 可移植算法 */
static unsigned long int next = 1;  /* 种子  */

unsigned int rand0(void)
{
    /* 生成伪随机数的魔术公式 */
    next = next * 1103515245 + 12345;
    return (unsigned int) (next / 65536) % 32768;
}
```

在程序清单 12.7 中，静态变量 next 的初始值是 1，其值在每次调用 rand0() 函数时都会被修改（通过魔术公式）。该函数是用于返回一个 0～32767 之间的值。注意，next 是具有内部链接的静态变量（并非无链接）。这是为了方便稍后扩展本例，供同一个文件中的其他函数共享。

程序清单 12.8 是测试 rand0() 函数的一个简单的驱动程序。

程序清单 12.8　r_drive0.c 驱动程序

```
/* r_drive0.c -- 测试 rand0()函数 */
/* 与 rand0.c 一起编译*/
#include <stdio.h>
extern unsigned int rand0(void);

int main(void)
{
    int count;

    for (count = 0; count < 5; count++)
        printf("%d\n", rand0());
```

```
    return 0;
}
```

该程序也需要多文件编译。程序清单 12.7 和程序清单 12.8 分别使用一个文件。程序清单 12.8 中的 extern 关键字提醒读者 rand0()被定义在其他文件中，在这个文件中不要求写出该函数原型。输出如下：

```
16838
5758
10113
17515
31051
```

程序输出的数字看上去是随机的，再次运行程序后，输出如下：

```
16838
5758
10113
17515
31051
```

看来，这两次的输出完全相同，这体现了“伪随机”的一个方面。每次主程序运行，都开始于相同的种子 1。可以引入另一个函数 srand1()重置种子来解决这个问题。关键是要让 next 成为只供 rand1()和 srand1()访问的内部链接静态变量（srand1()相当于 C 库中的 srand()函数）。把 srand1()加入 rand1()所在的文件中。程序清单 12.9 给出了修改后的文件。

程序清单 12.9　s_and_r.c 文件程序

```
/* s_and_r.c -- 包含 rand1() 和 srand1() 的文件    */
/*              使用 ANSI C 可移植算法      */
static unsigned long int next = 1;  /* 种子 */

int rand1(void)
{
    /*生成伪随机数的魔术公式*/
    next = next * 1103515245 + 12345;
    return (unsigned int) (next / 65536) % 32768;
}

void srand1(unsigned int seed)
{
    next = seed;
}
```

注意，next 是具有内部链接的文件作用域静态变量。这意味着 rand1()和 srand1()都可以使用它，但是其他文件中的函数无法访问它。使用程序清单 12.10 的驱动程序测试这两个函数。

程序清单 12.10　r_drive1.c 驱动程序

```
/* r_drive1.c -- 测试 rand1() 和 srand1() */
/* 与 s_and_r.c 一起编译 */
#include <stdio.h>
#include <stdlib.h>
extern void srand1(unsigned int x);
extern int rand1(void);
```

```
int main(void)
{
    int count;
    unsigned seed;

    printf("Please enter your choice for seed.\n");
    while (scanf("%u", &seed) == 1)
    {
        srand1(seed);    /* 重置种子 */
        for (count = 0; count < 5; count++)
            printf("%d\n", rand1());
        printf("Please enter next seed (q to quit):\n");
    }
    printf("Done\n");

    return 0;
}
```

编译两个文件，运行该程序后，其输出如下：

```
1
16838
5758
10113
17515
31051
Please enter next seed (q to quit):
513
20067
23475
8955
20841
15324
Please enter next seed (q to quit):
q
Done
```

设置 seed 的值为 1，输出的结果与前面程序相同。但是设置 seed 的值为 513 后就得到了新的结果。

注意　自动重置种子

如果 C 实现允许访问一些可变的量（如，时钟系统），可以用这些值（可能会被截断）初始化种子值。例如，ANSI C 有一个 time() 函数返回系统时间。虽然时间单元因系统而异，但是重点是该返回值是一个可进行运算的类型，而且其值随着时间变化而变化。time() 返回值的类型名是 time_t，具体类型与系统有关。这没关系，我们可以使用强制类型转换：

```
#include <time.h> /* 提供 time()的 ANSI 原型*/
srand1((unsigned int) time(0)); /* 初始化种子 */
```

一般而言，time() 接受的参数是一个 time_t 类型对象的地址，而时间值就储存在传入的地址上。当然，也可以传入空指针（0）作为参数，这种情况下，只能通过返回值机制来提供值。

可以把这个技巧应用于标准的 ANSI C 函数 srand() 和 rand() 中。如果使用这些函数，要在文件中包含 stdlib.c 头文件。实际上，既然已经明白了 srand1() 和 rand1() 如何使用内部链接的静态变量，

你也可以使用编译器提供的版本。我们将在下一个示例中这样做。

12.3　掷骰子

我们将要模拟一个非常流行的游戏——掷骰子。骰子的形式多种多样，最普遍的是使用两个 6 面骰子。在一些冒险游戏中，会使用 5 种骰子：4 面、6 面、8 面、12 面和 20 面。聪明的古希腊人证明了只有 5 种正多面体，它们的所有面都具有相同的形状和大小。各种不同类型的骰子就是根据这些正多面体发展而来。也可以做成其他面数的，但是其所有的面不会都相等，因此各个面朝上的几率就不同。

计算机计算不用考虑几何的限制，所以可以设计任意面数的电子骰子。我们先从 6 面开始。

我们想获得 1～6 的随机数。然而，rand()生成的随机数在 0～RAND_MAX 之间。RAND_MAX 被定义在 stdlib.h 中，其值通常是 INT_MAX。因此，需要进行一些调整，方法如下。

1．把随机数求模 6，获得的整数在 0～5 之间。

2．结果加 1，新值在 1～6 之间。

3．为方便以后扩展，把第 1 步中的数字 6 替换成骰子面数。

下面的代码实现了这 3 个步骤：

```
#include <stdlib.h> /* 提供 rand()的原型 */
int rollem(int sides)
{
    int roll;

    roll = rand() % sides + 1;
    return roll;
}
```

我们还想用一个函数提示用户选择任意面数的骰子，并返回点数总和。如程序清单 12.11 所示。

程序清单 12.11　diceroll.c 程序

```
/* diceroll.c -- 掷骰子模拟程序 */
                               /* 与 mandydice.c 一起编译 */
#include "diceroll.h"
#include <stdio.h>
#include <stdlib.h>            /* 提供库函数 rand()的原型 */

int roll_count = 0;            /* 外部链接 */

static int rollem(int sides)   /* 该函数属于该文件私有 */
{
    int roll;

    roll = rand() % sides + 1;
    ++roll_count;              /* 计算函数调用次数 */

    return roll;
}

int roll_n_dice(int dice, int sides)
{
    int d;
```

```
    int total = 0;
    if (sides < 2)
    {
        printf("Need at least 2 sides.\n");
        return -2;
    }
    if (dice < 1)
    {
        printf("Need at least 1 die.\n");
        return -1;
    }

    for (d = 0; d < dice; d++)
        total += rollem(sides);

    return total;
}
```

该文件加入了新元素。第一，rollem()函数属于该文件私有，它是 roll_n_dice()的辅助函数。第二，为了演示外部链接的特性，该文件声明了一个外部变量 roll_count。该变量统计调用 rollem()函数的次数。这样设计有点蹩脚，仅为了演示外部变量的特性。第三，该文件包含以下预处理指令：

```
#include "diceroll.h"
```

如果使用标准库函数，如 rand()，要在当前文件中包含标准头文件（对 rand()而言要包含 stdlib.h），而不是声明该函数。因为头文件中已经包含了正确的函数原型。我们效仿这一做法，把 roll_n_dice()函数的原型放在 diceroll.h 头文件中。把文件名放在双引号中而不是尖括号中，指示编译器在本地查找文件，而不是到编译器存放标准头文件的位置去查找文件。“本地查找”的含义取决于具体的实现。一些常见的实现把头文件与源代码文件或工程文件（如果编译器使用它们的话）放在相同的目录或文件夹中。程序清单 12.12 是头文件中的内容。

程序清单 12.12　diceroll.h 文件

```
//diceroll.h
extern int roll_count;

int roll_n_dice(int dice, int sides);
```

该头文件中包含一个函数原型和一个 extern 声明。由于 direroll.c 文件包含了该文件，direroll.c 实际上包含了 roll_count 的两个声明：

```
extern int roll_count;      // 头文件中的声明（引用式声明）
int roll_count = 0;         // 源代码文件中的声明（定义式声明）
```

这样做没问题。一个变量只能有一个定义式声明，但是带 extern 的声明是引用式声明，可以有多个引用式声明。

使用 roll_n_dice()函数的程序都要包含 diceroll.c 头文件。包含该头文件后，程序便可使用 roll_n_dice()函数和 roll_count 变量。如程序清单 12.13 所示。

程序清单 12.13　manydice.c 文件

```
/* manydice.c -- 多次掷骰子的模拟程序 */
/* 与 diceroll.c 一起编译*/
#include <stdio.h>
#include <stdlib.h>         /* 为库函数 srand() 提供原型  */
```

```
#include <time.h>           /* 为 time() 提供原型              */
#include "diceroll.h"      /* 为roll_n_dice()提供原型，为roll_count变量提供声明 */

int main(void)
{
    int dice, roll;
    int sides;
    int status;

    srand((unsigned int) time(0)); /* 随机种子 */
    printf("Enter the number of sides per die, 0 to stop.\n");
    while (scanf("%d", &sides) == 1 && sides > 0)
    {
        printf("How many dice?\n");
        if ((status = scanf("%d", &dice)) != 1)
        {
            if (status == EOF)
                break;             /* 退出循环 */
            else
            {
                printf("You should have entered an integer.");
                printf(" Let's begin again.\n");
                while (getchar() != '\n')
                    continue;      /* 处理错误的输入 */
             printf("How many sides? Enter 0 to stop.\n");
             continue;             /* 进入循环的下一轮迭代 */
            }
        }
        roll = roll_n_dice(dice, sides);
        printf("You have rolled a %d using %d %d-sided dice.\n",
            roll, dice, sides);
        printf("How many sides? Enter 0 to stop.\n");
    }
    printf("The rollem() function was called %d times.\n",
        roll_count);          /* 使用外部变量 */

    printf("GOOD FORTUNE TO YOU!\n");

    return 0;
}
```

要与包含程序清单 12.11 的文件一起编译该文件。可以把程序清单 12.11、12.12 和 12.13 都放在同一文件夹或目录中。运行该程序，下面是一个输出示例：

```
Enter the number of sides per die, 0 to stop.
6
How many dice?
2
You have rolled a 12 using 2 6-sided dice.
How many sides? Enter 0 to stop.
6
How many dice?
2
You have rolled a 4 using 2 6-sided dice.
```

```
How many sides? Enter 0 to stop.
6
How many dice?
2
You have rolled a 5 using 2 6-sided dice.
How many sides? Enter 0 to stop.
0
The rollem() function was called 6 times.
GOOD FORTUNE TO YOU!
```

因为该程序使用了 srand()随机生成随机数种子，所以大多数情况下，即使输入相同也很难得到相同的输出。注意，manydice.c 中的 main()访问了定义在 diceroll.c 中的 roll_count 变量。

有 3 种情况可以导致外层 while 循环结束：side 小于 1、输入类型不匹配（此时 scanf()返回 0）、遇到文件结尾（返回值是 EOF）。为了读取骰子的点数，该程序处理文件结尾的方式（退出 while 循环）与处理类型不匹配（进入循环的下一轮迭代）的情况不同。

可以通过多种方式使用 roll_n_dice()。sides 等于 2 时，程序模仿掷硬币，“正面朝上”为 2，“反面朝上”为 1（或者反过来表示也行）。很容易修改该程序单独显示点数的结果，或者构建一个骰子模拟器。如果要掷多次骰子（如在一些角色扮演类游戏中），可以很容易地修改程序以输出类似的结果：

```
Enter the number of sets; enter q to stop.
18
How many sides and how many dice?
6 3
Here are 18 sets of 3 6-sided throws.
12 10 6 9 8 14 8 15 9 14 12 17 11 7 10
13 8 14
How many sets? Enter q to stop.
q
```

rand1()或 rand()（不是 rollem()）还可以用来创建一个猜数字程序，让计算机选定一个数字，你来猜。读者感兴趣的话可以自己编写这个程序。

12.4　分配内存：malloc()和 free()

我们前面讨论的存储类别有一个共同之处：在确定用哪种存储类别后，根据已制定好的内存管理规则，将自动选择其作用域和存储期。然而，还有更灵活地选择，即用库函数分配和管理内存。

首先，回顾一下内存分配。所有程序都必须预留足够的内存来储存程序使用的数据。这些内存中有些是自动分配的。例如，以下声明：

```
float x;
char place[] = "Dancing Oxen Creek";
```

为一个 float 类型的值和一个字符串预留了足够的内存，或者可以显式指定分配一定数量的内存：

```
int plates[100];
```

该声明预留了 100 个内存位置，每个位置都用于储存 int 类型的值。声明还为内存提供了一个标识符。因此，可以使用 x 或 place 识别数据。回忆一下，静态数据在程序载入内存时分配，而自动数据在程序执行块时分配，并在程序离开该块时销毁。

C 能做的不止这些。可以在程序运行时分配更多的内存。主要的工具是 malloc()函数，该函数接受一个参数：所需的内存字节数。malloc()函数会找到合适的空闲内存块，这样的内存是匿名的。也就是说，malloc()分配内存，但是不会为其赋名。然而，它确实返回动态分配内存块的首字节地址。因此，可以把该地址赋给一个指针变量，并使用指针访问这块内存。因为 char 表示 1 字节，malloc()的返回类型通常

被定义为指向 char 的指针。然而，从 ANSI C 标准开始，C 使用一个新的类型：指向 void 的指针。该类型相当于一个“通用指针”。malloc()函数可用于返回指向数组的指针、指向结构的指针等，所以通常该函数的返回值会被强制转换为匹配的类型。在 ANSI C 中，应该坚持使用强制类型转换，提高代码的可读性。然而，把指向 void 的指针赋给任意类型的指针完全不用考虑类型匹配的问题。如果 malloc()分配内存失败，将返回空指针。

我们试着用 malloc()创建一个数组。除了用 malloc()在程序运行时请求一块内存，还需要一个指针记录这块内存的位置。例如，考虑下面的代码：

```
double * ptd;
ptd = (double *) malloc(30 * sizeof(double));
```

以上代码为 30 个 double 类型的值请求内存空间，并设置 ptd 指向该位置。注意，指针 ptd 被声明为指向一个 double 类型，而不是指向内含 30 个 double 类型值的块。回忆一下，数组名是该数组首元素的地址。因此，如果让 ptd 指向这个块的首元素，便可像使用数组名一样使用它。也就是说，可以使用表达式 ptd[0]访问该块的首元素，ptd[1]访问第 2 个元素，以此类推。根据前面所学的知识，可以使用数组名来表示指针，也可以用指针来表示数组。

现在，我们有 3 种创建数组的方法。

- 声明数组时，用常量表达式表示数组的维度，用数组名访问数组的元素。可以用静态内存或自动内存创建这种数组。
- 声明变长数组（C99 新增的特性）时，用变量表达式表示数组的维度，用数组名访问数组的元素。具有这种特性的数组只能在自动内存中创建。
- 声明一个指针，调用 malloc()，将其返回值赋给指针，使用指针访问数组的元素。该指针可以是静态的或自动的。

使用第 2 种和第 3 种方法可以创建*动态数组*（*dynamic array*）。这种数组和普通数组不同，可以在程序运行时选择数组的大小和分配内存。例如，假设 n 是一个整型变量。在 C99 之前，不能这样做：

```
double item[n]; /* C99之前：n不允许是变量 */
```

但是，可以这样做：

```
ptd = (double *) malloc(n * sizeof(double)); /* 可以 */
```

如你所见，这比变长数组更灵活。

通常，malloc()要与 free()配套使用。free()函数的参数是之前 malloc()返回的地址，该函数释放之前 malloc()分配的内存。因此，动态分配内存的存储期从调用 malloc()分配内存到调用 free()释放内存为止。设想 malloc()和 free()管理着一个内存池。每次调用 malloc()分配内存给程序使用，每次调用 free()把内存归还内存池中，这样便可重复使用这些内存。free()的参数应该是一个指针，指向由 malloc()分配的一块内存。不能用 free()释放通过其他方式（如，声明一个数组）分配的内存。malloc()和 free()的原型都在 stdlib.h 头文件中。

使用 malloc()，程序可以在运行时才确定数组大小。如程序清单 12.14 所示，它把内存块的地址赋给指针 ptd，然后便可以使用数组名的方式使用 ptd。另外，如果内存分配失败，可以调用 exit()函数结束程序，其原型在 stdlib.h 中。EXIT_FAILURE 的值也被定义在 stdlib.h 中。标准提供了两个返回值以保证在所有操作系统中都能正常工作：EXIT_SUCCESS（或者，相当于 0）表示普通的程序结束，EXIT_FAILURE 表示程序异常中止。一些操作系统（包括 UNIX、Linux 和 Windows）还接受一些表示其他运行错误的整数值。

程序清单 12.14　dyn_arr.c **程序**

```
/* dyn_arr.c -- 动态分配数组 */
#include <stdio.h>
#include <stdlib.h> /* 为 malloc()、free()提供原型 */

int main(void)
{
    double * ptd;
    int max;
    int number;
    int i = 0;

    puts("What is the maximum number of type double entries?");
    if (scanf("%d", &max) != 1)
    {
        puts("Number not correctly entered -- bye.");
        exit(EXIT_FAILURE);
    }
    ptd = (double *) malloc(max * sizeof(double));
    if (ptd == NULL)
    {
        puts("Memory allocation failed. Goodbye.");
        exit(EXIT_FAILURE);
    }
    /* ptd 现在指向有 max 个元素的数组 */
    puts("Enter the values (q to quit):");
    while (i < max && scanf("%lf", &ptd[i]) == 1)
        ++i;
    printf("Here are your %d entries:\n", number = i);
    for (i = 0; i < number; i++)
    {
        printf("%7.2f ", ptd[i]);
        if (i % 7 == 6)
            putchar('\n');
    }
    if (i % 7 != 0)
        putchar('\n');
    puts("Done.");
    free(ptd);

    return 0;
}
```

下面是该程序的运行示例。程序通过交互的方式让用户先确定数组的大小，我们设置数组大小为 5。虽然我们后来输入了 6 个数，但程序也只处理前 5 个数。

```
What is the maximum number of entries?
5
Enter the values (q to quit):
20 30 35 25 40 80
Here are your 5 entries:
  20.00 30.00 35.00 25.00 40.00
Done.
```

该程序通过以下代码获取数组的大小：

```
if (scanf("%d", &max) != 1)
{
    puts("Number not correctly entered -- bye.");
    exit(EXIT_FAILURE);
}
```

接下来，分配足够的内存空间以储存用户要存入的所有数，然后把动态分配的内存地址赋给指针 ptd：

```
ptd = (double *) malloc(max * sizeof (double));
```

在 C 中，不一定要使用强制类型转换(double *)，但是在 C++中必须使用。所以，使用强制类型转换更容易把 C 程序转换为 C++程序。

malloc()可能分配不到所需的内存。在这种情况下，该函数返回空指针，程序结束：

```
if (ptd == NULL)
{
    puts("Memory allocation failed. Goodbye.");
    exit(EXIT_FAILURE);
}
```

如果程序成功分配内存，便可把 ptd 视为一个有 max 个元素的数组名。

注意，free()函数位于程序的末尾，它释放了 malloc()函数分配的内存。free()函数只释放其参数指向的内存块。一些操作系统在程序结束时会自动释放动态分配的内存，但是有些系统不会。为保险起见，请使用 free()，不要依赖操作系统来清理。

使用动态数组有什么好处？从本例来看，使用动态数组给程序带来了更多灵活性。假设你已经知道，在大多数情况下程序所用的数组都不会超过 100 个元素，但是有时程序确实需要 10000 个元素。要是按照平时的做法，你不得不为这种情况声明一个内含 10000 个元素的数组。基本上这样做是在浪费内存。如果需要 10001 个元素，该程序就会出错。这种情况下，可以使用一个动态数组调整程序以适应不同的情况。

12.4.1 free()的重要性

静态内存的数量在编译时是固定的，在程序运行期间也不会改变。自动变量使用的内存数量在程序执行期间自动增加或减少。但是动态分配的内存数量只会增加，除非用 free()进行释放。例如，假设有一个创建数组临时副本的函数，其代码框架如下：

```
...
int main()
{
    double glad[2000];
    int i;
    ...
    for (i = 0; i < 1000; i++)
        gobble(glad, 2000);
    ...
}
void gobble(double ar[], int n)
{
    double * temp = (double *) malloc( n * sizeof(double));
    ... /* free(temp); // 假设忘记使用 free() */
}
```

第 1 次调用 gobble()时，它创建了指针 temp，并调用 malloc()分配了 16000 字节的内存（假设 double 为 8 字节）。假设如代码注释所示，遗漏了 free()。当函数结束时，作为自动变量的指针 temp 也会消失。但是它所指向的 16000 字节的内存却仍然存在。由于 temp 指针已被销毁，所以无法访问这块

内存，它也不能被重复使用，因为代码中没有调用 free()释放这块内存。

第 2 次调用 gobble()时，它又创建了指针 temp，并调用 malloc()分配了 16000 字节的内存。第 1 次分配的 16000 字节内存已不可用，所以 malloc()分配了另外一块 16000 字节的内存。当函数结束时，该内存块也无法被再访问和再使用。

循环要执行 1000 次，所以在循环结束时，内存池中有 1600 万字节被占用。实际上，也许在循环结束之前就已耗尽所有的内存。这类问题被称为*内存泄漏*（*memory leak*）。在函数末尾处调用 free()函数可避免这类问题发生。

12.4.2　calloc()函数

分配内存还可以使用 calloc()，典型的用法如下：

```
long * newmem;
newmem = (long *)calloc(100, sizeof (long));
```

和 malloc()类似，在 ANSI 之前，calloc()也返回指向 char 的指针；在 ANSI 之后，返回指向 void 的指针。如果要储存不同的类型，应使用强制类型转换运算符。calloc()函数接受两个无符号整数作为参数（ANSI 规定是 size_t 类型）。第 1 个参数是所需的存储单元数量，第 2 个参数是存储单元的大小（以字节为单位）。在该例中，long 为 4 字节，所以，前面的代码创建了 100 个 4 字节的存储单元，总共 400 字节。

用 sizeof(long)而不是 4，提高了代码的可移植性。这样，在其他 long 不是 4 字节的系统中也能正常工作。

calloc()函数还有一个特性：它把块中的所有位都设置为 0（注意，在某些硬件系统中，不是把所有位都设置为 0 来表示浮点值 0）。

free()函数也可用于释放 calloc()分配的内存。

动态内存分配是许多高级程序设计技巧的关键。我们将在第 17 章中详细讲解。有些编译器可能还提供其他内存管理函数，有些可以移植，有些不可以。读者可以抽时间看一下。

12.4.3　动态内存分配和变长数组

变长数组（VLA）和调用 malloc()在功能上有些重合。例如，两者都可用于创建在运行时确定大小的数组：

```
int vlamal()
{
    int n;
    int * pi;
    scanf("%d", &n);
    pi = (int *) malloc (n * sizeof(int));
    int ar[n];// 变长数组
    pi[2] = ar[2] = -5;
    ...
}
```

不同的是，变长数组是自动存储类型。因此，程序在离开变长数组定义所在的块时（该例中，即 vlamal()函数结束时），变长数组占用的内存空间会被自动释放，不必使用 free()。另一方面，用 malloc()创建的数组不必局限在一个函数内访问。例如，可以这样做：被调函数创建一个数组并返回指针，供主调函数访问，然后主调函数在末尾调用 free()释放之前被调函数分配的内存。另外，free()

所用的指针变量可以与 malloc()的指针变量不同，但是两个指针必须储存相同的地址。但是，不能释放同一块内存两次。

对多维数组而言，使用变长数组更方便。当然，也可以用 malloc()创建二维数组，但是语法比较繁琐。如果编译器不支持变长数组特性，就只能固定二维数组的维度，如下所示：

```
int n = 5;
int m = 6;
int ar2[n][m]; // n×m的变长数组（VLA）
int (* p2)[6]; // C99之前的写法
int (* p3)[m]; // 要求支持变长数组
p2 = (int (*)[6]) malloc(n * 6 * sizeof(int)); // n×6 数组
p3 = (int (*)[m]) malloc(n * m * sizeof(int)); // n×m 数组（要求支持变长数组）
ar2[1][2] = p2[1][2] = 12;
```

先复习一下指针声明。由于 malloc()函数返回一个指针，所以 p2 必须是一个指向合适类型的指针。第 1 个指针声明：

```
int (* p2)[6]; // C99之前的写法
```

表明 p2 指向一个内含 6 个 int 类型值的数组。因此，p2[i]代表一个由 6 个整数构成的元素，p2[i][j]代表一个整数。

第 2 个指针声明用一个变量指定 p3 所指向数组的大小。因此，p3 代表一个指向变长数组的指针，这行代码不能在 C90 标准中运行。

12.4.4 存储类别和动态内存分配

存储类别和动态内存分配有何联系？我们来看一个理想化模型。可以认为程序把它可用的内存分为 3 部分：一部分供具有外部链接、内部链接和无链接的静态变量使用；一部分供自动变量使用；一部分供动态内存分配。

静态存储类别所用的内存数量在编译时确定，只要程序还在运行，就可访问储存在该部分的数据。该类别的变量在程序开始执行时被创建，在程序结束时被销毁。

然而，自动存储类别的变量在程序进入变量定义所在块时存在，在程序离开块时消失。因此，随着程序调用函数和函数结束，自动变量所用的内存数量也相应地增加和减少。这部分的内存通常作为栈来处理，这意味着新创建的变量按顺序加入内存，然后以相反的顺序销毁。

动态分配的内存在调用 malloc()或相关函数时存在，在调用 free()后释放。这部分的内存由程序员管理，而不是一套规则。所以内存块可以在一个函数中创建，在另一个函数中销毁。正是因为这样，这部分的内存用于动态内存分配会支离破碎。也就是说，未使用的内存块分散在已使用的内存块之间。另外，使用动态内存通常比使用栈内存慢。

总而言之，程序把静态对象、自动对象和动态分配的对象储存在不同的区域。

程序清单 12.15 where.c 程序

```
//  where.c  -- 数据被储存在何处?

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int static_store = 30;
const char * pcg = "String Literal";
```

```
int main()
{
    int auto_store = 40;
    char auto_string [] = "Auto char Array";
    int * pi;
    char * pcl;

    pi = (int *) malloc(sizeof(int));
    *pi = 35;
    pcl = (char *) malloc(strlen("Dynamic String") + 1);
    strcpy(pcl, "Dynamic String");

    printf("static_store: %d at %p\n", static_store, &static_store);
    printf("  auto_store: %d at %p\n", auto_store, &auto_store);
    printf("         *pi: %d at %p\n", *pi, pi);
    printf("  %s at %p\n", pcg, pcg);
    printf(" %s at %p\n", auto_string, auto_string);
    printf("  %s at %p\n", pcl, pcl);
    printf("   %s at %p\n", "Quoted String", "Quoted String");
    free(pi);
    free(pcl);

    return 0;
}
```

在我们的系统中，该程序的输入如下：

```
static_store: 30 at 00378000
  auto_store: 40 at 0049FB8C
         *pi: 35 at 008E9BA0
  String Literal at 00375858
 Auto char Array at 0049FB74
  Dynamic String at 008E9BD0
   Quoted String at 00375908
```

如上所示，静态数据（包括字符串字面量）占用一个区域，自动数据占用另一个区域，动态分配的数据占用第 3 个区域（通常被称为内存堆或自由内存）。

12.5　ANSI C 类型限定符

我们通常用类型和存储类别来描述一个变量。C90 还新增了两个属性：恒常性（*constancy*）和易变性（*volatility*）。这两个属性可以分别用关键字 `const` 和 `volatile` 来声明，以这两个关键字创建的类型是限定类型（*qualified type*）。C99 标准新增了第 3 个限定符：`restrict`，用于提高编译器优化。C11 标准新增了第 4 个限定符：`_Atomic`。C11 提供一个可选库，由 `stdatomic.h` 管理，以支持并发程序设计，而且`_Atomic` 是可选支持项。

C99 为类型限定符增加了一个新属性：它们现在是幂等的（*idempotent*）！这个属性听起来很强大，其实意思是可以在一条声明中多次使用同一个限定符，多余的限定符将被忽略：

```
const const const int n = 6; // 与 const int n = 6;相同
```

有了这个新属性，就可以编写类似下面的代码：

```
typedef const int zip;
const zip q = 8;
```

12.5.1 const 类型限定符

第 4 章和第 10 章中介绍过 const。以 const 关键字声明的对象，其值不能通过赋值或递增、递减来修改。在 ANSI 兼容的编译器中，以下代码：

```
const int nochange;    /* 限定 nochange 的值不能被修改 */
nochange = 12;         /* 不允许 */
```

编译器会报错。但是，可以初始化 const 变量。因此，下面的代码没问题：

```
const int nochange = 12; /* 没问题 */
```

该声明让 nochange 成为只读变量。初始化后，就不能再改变它的值。

可以用 const 关键字创建不允许修改的数组：

```
const int days1[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
```

1. 在指针和形参声明中使用 const

声明普通变量和数组时使用 const 关键字很简单。指针则复杂一些，因为要区分是限定指针本身为 const 还是限定指针指向的值为 const。下面的声明：

```
const float * pf; /* pf 指向一个 float 类型的 const 值 */
```

创建了 pf 指向的值不能被改变，而 pt 本身的值可以改变。例如，可以设置该指针指向其他 const 值。相比之下，下面的声明：

```
float * const pt; /* pt 是一个 const 指针 */
```

创建的指针 pt 本身的值不能更改。pt 必须指向同一个地址，但是它所指向的值可以改变。下面的声明：

```
const float * const ptr;
```

表明 ptr 既不能指向别处，它所指向的值也不能改变。

还可以把 const 放在第 3 个位置：

```
float const * pfc; // 与 const float * pfc;相同
```

如注释所示，把 const 放在类型名之后、*之前，说明该指针不能用于改变它所指向的值。简而言之，const 放在*左侧任意位置，限定了指针指向的数据不能改变；const 放在*的右侧，限定了指针本身不能改变。

const 关键字的常见用法是声明为函数形参的指针。例如，假设有一个函数要调用 display()显示一个数组的内容。要把数组名作为实际参数传递给该函数，但是数组名是一个地址。该函数可能会更改主调函数中的数据，但是下面的原型保证了数据不会被更改：

```
void display(const int array[], int limit);
```

在函数原型和函数头，形参声明 const int array[]与 const int * array 相同，所以该声明表明不能更改 array 指向的数据。

ANSI C 库遵循这种做法。如果一个指针仅用于给函数访问值，应将其声明为一个指向 const 限定类型的指针。如果要用指针更改主调函数中的数据，就不使用 const 关键字。例如，ANSI C 中的 strcat()原型如下：

```
char *strcat(char * restrict s1, const char * restrict s2);
```

回忆一下，strcat()函数在第 1 个字符串的末尾添加第 2 个字符串的副本。这更改了第 1 个字符串，但是未更改第 1 个字符串。上面的声明体现了这一点。

2. 对全局数据使用 const

前面讲过，使用全局变量是一种冒险的方法，因为这样做暴露了数据，程序的任何部分都能更改数据。如果把数据设置为 const，就可避免这样的危险，因此用 const 限定符声明全局数据很合理。可以创建 const 变量、const 数组和 const 结构（结构是一种复合数据类型，将在下一章介绍）。

然而，在文件间共享 const 数据要小心。可以采用两个策略。第一，遵循外部变量的常用规则，即在一个文件中使用定义式声明，在其他文件中使用引用式声明（用 extern 关键字）：

```
/* file1.c -- 定义了一些外部 const 变量 */
const double PI = 3.14159;
const char * MONTHS[12] = { "January", "February", "March", "April", "May",
                            "June", "July","August", "September", "October",
                            "November", "December" };

/* file2.c -- 使用定义在别处的外部 const 变量 */
extern const double PI;
extern const * MONTHS [];
```

另一种方案是，把 const 变量放在一个头文件中，然后在其他文件中包含该头文件：

```
/* constant.h --定义了一些外部 const 变量*/
static const double PI = 3.14159;
static const char * MONTHS[12] ={"January", "February", "March", "April", "May",
                                 "June", "July","August", "September", "October",
                                 "November", "December"};

/* file1.c --使用定义在别处的外部 const 变量*/
#include "constant.h"

/* file2.c --使用定义在别处的外部 const 变量*/
#include "constant.h"
```

这种方案必须在头文件中用关键字 static 声明全局 const 变量。如果去掉 static，那么在 file1.c 和 file2.c 中包含 constant.h 将导致每个文件中都有一个相同标识符的定义式声明，C 标准不允许这样做（然而，有些编译器允许）。实际上，这种方案相当于给每个文件提供了一个单独的数据副本[1]。由于每个副本只对该文件可见，所以无法用这些数据和其他文件通信。不过没关系，它们都是完全相同（每个文件都包含相同的头文件）的 const 数据（声明时使用了 const 关键字），这不是问题。

头文件方案的好处是，方便你偷懒，不用惦记着在一个文件中使用定义式声明，在其他文件中使用引用式声明。所有的文件都只需包含同一个头文件即可。但它的缺点是，数据是重复的。对于前面的例子而言，这不算什么问题，但是如果 const 数据包含庞大的数组，就不能视而不见了。

12.5.2 volatile 类型限定符

volatile 限定符告知计算机，代理（而不是变量所在的程序）可以改变该变量的值。通常，它被用于硬件地址以及在其他程序或同时运行的线程中共享数据。例如，一个地址上可能储存着当前的时钟时间，无论程序做什么，地址上的值都随时间的变化而改变。或者一个地址用于接受另一台计算机传入的信息。

volatile 的语法和 const 一样：

```
volatile int loc1;/* loc1 是一个易变的位置 */
volatile int * ploc;   /* ploc 是一个指向易变的位置的指针 */
```

[1] 注意，以 static 声明的文件作用域变量具有内部链接属性。——译者注

以上代码把 loc1 声明为 volatile 变量，把 ploc 声明为指向 volatile 变量的指针。

读者可能认为 volatile 是个可有可无的概念，为何 ANSI 委员把 volatile 关键字放入标准？原因是它涉及编译器的优化。例如，假设有下面的代码：

```
val1 = x;
/* 一些不使用 x 的代码*/
val2 = x;
```

智能的（进行优化的）编译器会注意到以上代码使用了两次 x，但并未改变它的值。于是编译器把 x 的值临时储存在寄存器中，然后在 val2 需要使用 x 时，才从寄存器中（而不是从原始内存位置上）读取 x 的值，以节约时间。这个过程被称为高速缓存（*caching*）。通常，高速缓存是个不错的优化方案，但是如果一些其他代理在以上两条语句之间改变了 x 的值，就不能这样优化了。如果没有 volatile 关键字，编译器就不知道这种事情是否会发生。因此，为安全起见，编译器不会进行高速缓存。这是在 ANSI 之前的情况。现在，如果声明中没有 volatile 关键字，编译器会假定变量的值在使用过程中不变，然后再尝试优化代码。

可以同时用 const 和 volatile 限定一个值。例如，通常用 const 把硬件时钟设置为程序不能更改的变量，但是可以通过代理改变，这时用 volatile。只能在声明中同时使用这两个限定符，它们的顺序不重要，如下所示：

```
volatile const int loc;
const volatile int * ploc;
```

12.5.3 restrict 类型限定符

restrict 关键字允许编译器优化某部分代码以更好地支持计算。它只能用于指针，表明该指针是访问数据对象的唯一且初始的方式。要弄明白为什么这样做有用，先看几个例子。考虑下面的代码：

```
int ar[10];
int * restrict restar = (int *) malloc(10 * sizeof(int));
int * par = ar;
```

这里，指针 restar 是访问由 malloc()所分配内存的唯一且初始的方式。因此，可以用 restrict 关键字限定它。而指针 par 既不是访问 ar 数组中数据的初始方式，也不是唯一方式。所以不用把它设置为 restrict。

现在考虑下面稍复杂的例子，其中 n 是 int 类型：

```
for (n = 0; n < 10; n++)
{
    par[n] += 5;
    restar[n] += 5;
    ar[n] *= 2;
    par[n] += 3;
    restar[n] += 3;
}
```

由于之前声明了 restar 是访问它所指向的数据块的唯一且初始的方式，编译器可以把涉及 restar 的两条语句替换成下面这条语句，效果相同：

```
restar[n] += 8; /* 可以进行替换 */
```

但是，如果把与 par 相关的两条语句替换成下面的语句，将导致计算错误：

```
par[n] += 8; / * 给出错误的结果 */
```

这是因为 for 循环在 par 两次访问相同的数据之间，用 ar 改变了该数据的值。

在本例中，如果未使用 restrict 关键字，编译器就必须假设最坏的情况（即，在两次使用指针之间，其他的标识符可能已经改变了数据）。如果用了 restrict 关键字，编译器就可以选择捷径优化计算。

restrict 限定符还可用于函数形参中的指针。这意味着编译器可以假定在函数体内其他标识符不会修改该指针指向的数据，而且编译器可以尝试对其优化，使其不做别的用途。例如，C 库有两个函数用于把一个位置上的字节拷贝到另一个位置。在 C99 中，这两个函数的原型是：

```
void * memcpy(void * restrict s1, const void * restrict s2, size_t n);
void * memmove(void * s1, const void * s2, size_t n);
```

这两个函数都从位置 s2 把 n 字节拷贝到位置 s1。memcpy()函数要求两个位置不重叠，但是 memove()没有这样的要求。声明 s1 和 s2 为 restrict 说明这两个指针都是访问相应数据的唯一方式，所以它们不能访问相同块的数据。这满足了 memcpy()无重叠的要求。memmove()函数允许重叠，它在拷贝数据时不得不更小心，以防在使用数据之前就先覆盖了数据。

restrict 关键字有两个读者。一个是编译器，该关键字告知编译器可以自由假定一些优化方案。另一个读者是用户，该关键字告知用户要使用满足 restrict 要求的参数。总而言之，编译器不会检查用户是否遵循这一限制，但是无视它后果自负。

12.5.4 _Atomic 类型限定符（C11）

并发程序设计把程序执行分成可以同时执行的多个线程。这给程序设计带来了新的挑战，包括如何管理访问相同数据的不同线程。C11 通过包含可选的头文件 stdatomic.h 和 threads.h，提供了一些可选的（不是必须实现的）管理方法。值得注意的是，要通过各种宏函数来访问原子类型。当一个线程对一个原子类型的对象执行原子操作时，其他线程不能访问该对象。例如，下面的代码：

```
int hogs;// 普通声明
hogs = 12;    // 普通赋值
```

可以替换成：

```
_Atomic int hogs;           // hogs 是一个原子类型的变量
atomic_store(&hogs, 12);    // stdatomic.h 中的宏
```

这里，在 hogs 中储存 12 是一个原子过程，其他线程不能访问 hogs。

编写这种代码的前提是，编译器要支持这一新特性。

12.5.5 旧关键字的新位置

C99 允许把类型限定符和存储类别说明符 static 放在函数原型和函数头的形式参数的初始方括号中。对于类型限定符而言，这样做为现有功能提供了一个替代的语法。例如，下面是旧式语法的声明：

```
void ofmouth(int * const a1, int * restrict a2, int n);   // 以前的风格
```

该声明表明 a1 是一个指向 int 的 const 指针，这意味着不能更改指针本身，可以更改指针指向的数据。除此之外，还表明 a2 是一个 restrict 指针，如上一节所述。新的等价语法如下：

```
void ofmouth(int a1[const], int a2[restrict], int n);// C99 允许
```

根据新标准，在声明函数形参时，指针表示法和数组表示法都可以使用这两个限定符。

static 的情况不同，因为新标准为 static 引入了一种与以前用法不相关的新用法。现在，static 除了表明静态存储类别变量的作用域或链接外，新的用法告知编译器如何使用形式参数。例如，考虑下面的原型：

```
double stick(double ar[static 20]);
```

static 的这种用法表明，函数调用中的实际参数应该是一个指向数组首元素的指针，且该数组至少

有 20 个元素。这种用法的目的是让编译器使用这些信息优化函数的编码。为何给 static 新增一个完全不同的用法？C 标准委员会不愿意创建新的关键字，因为这样会让以前用新关键字作为标识符的程序无效。所以，他们会尽量利用现有的关键字，尽量不添加新的关键字。

restrict 关键字有两个读者。一个是编译器，该关键字告知编译器可以自由假定一些优化方案。另一个读者是用户，该关键字告知用户要使用满足 restrict 要求的参数。

12.6　关键概念

C 提供多种管理内存的模型。除了熟悉这些模型外，还要学会如何选择不同的类别。大多数情况下，最好选择自动变量。如果要使用其他类别，应该有充分的理由。通常，使用自动变量、函数形参和返回值进行函数间的通信比使用全局变量安全。但是，保持不变的数据适合用全局变量。

应该尽量理解静态内存、自动内存和动态分配内存的属性。尤其要注意：静态内存的数量在编译时确定；静态数据在载入程序时被载入内存。在程序运行时，自动变量被分配或释放，所以自动变量占用的内存数量随着程序的运行会不断变化。可以把自动内存看作是可重复利用的工作区。动态分配的内存也会增加和减少，但是这个过程由函数调用控制，不是自动进行的。

12.7　本章小结

内存用于存储程序中的数据，由存储期、作用域和链接表征。存储期可以是静态的、自动的或动态分配的。如果是静态存储期，在程序开始执行时分配内存，并在程序运行时都存在。如果是自动存储期，在程序进入变量定义所在块时分配变量的内存，在程序离开块时释放内存。如果是动态分配存储期，在调用 malloc()（或相关函数）时分配内存，在调用 free()函数时释放内存。

作用域决定程序的哪些部分可以访问某数据。定义在所有函数之外的变量具有文件作用域，对位于该变量声明之后的所有函数可见。定义在块或作为函数形参内的变量具有块作用域，只对该块以及它包含的嵌套块可见。

链接描述定义在程序某翻译单元中的变量可被链接的程度。具有块作用域的变量是局部变量，无链接。具有文件作用域的变量可以是内部链接或外部链接。内部链接意味着只有其定义所在的文件才能使用该变量。外部链接意味着其他文件使用也可以使用该变量。

下面是 C 的 5 种存储类别（不包括线程的概念）。

- **自动**——在块中不带存储类别说明符或带 auto 存储类别说明符声明的变量（或作为函数头中的形参）属于自动存储类别，具有自动存储期、块作用域、无链接。如果未初始化自动变量，它的值是未定义的。
- **寄存器**——在块中带 register 存储类别说明符声明的变量（或作为函数头中的形参）属于寄存器存储类别，具有自动存储期、块作用域、无链接，且无法获取其地址。把一个变量声明为寄存器变量即请求编译器将其储存到访问速度最快的区域。如果未初始化寄存器变量，它的值是未定义的。
- **静态、无链接**——在块中带 static 存储类别说明符声明的变量属于“静态、无链接”存储类别，具有静态存储期、块作用域、无链接。只在编译时被初始化一次。如果未显式初始化，它的字节都被设置为 0。
- **静态、外部链接**——在所有函数外部且没有使用 static 存储类别说明符声明的变量属于“静态、外部链接”存储类别，具有静态存储期、文件作用域、外部链接。只能在编译器被初始化一次。如果未显式初始化，它的字节都被设置为 0。
- **静态、内部链接**——在所有函数外部且使用了 static 存储类别说明符声明的变量属于“静态、内部链接”存储类别，具有静态存储期、文件作用域、内部链接。只能在编译器被初始化一次。如果未显式初始化，它的字节都被设置为 0。

动态分配的内存由 malloc()（或相关）函数分配，该函数返回一个指向指定字节数内存块的指针。这块内存被 free()函数释放后便可重复使用，free()函数以该内存块的地址作为参数。

类型限定符 const、volatile、restrict 和_Atomic。const 限定符限定数据在程序运行时不能改变。对指针使用 const 时，可限定指针本身不能改变或指针指向的数据不能改变，这取决于 const 在指针声明中的位置。volatile 限定符表明，限定的数据除了被当前程序修改外还可以被其他进程修改。该限定符的目的是警告编译器不要进行假定的优化。restrict 限定符也是为了方便编译器设置优化方案。restrict 限定的指针是访问它所指向数据的唯一途径。

12.8　复习题

复习题的参考答案在附录 A 中。

1．哪些类别的变量可以成为它所在函数的局部变量？

2．哪些类别的变量在它所在程序的运行期一直存在？

3．哪些类别的变量可以被多个文件使用？哪些类别的变量仅限于在一个文件中使用？

4．块作用域变量具有什么链接属性？

5．extern 关键字有什么用途？

6．考虑下面两行代码，就输出的结果而言有何异同：

```
int * p1 = (int *)malloc(100 * sizeof(int));
int * p1 = (int *)calloc(100, sizeof(int));
```

7．下面的变量对哪些函数可见？程序是否有误？

```
/* 文件 1 */
int daisy;
int main(void)
{
    int lily;
    ...;
}
int petal()
{
    extern int daisy, lily;
    ...;
}
/* 文件 2 */
extern int daisy;
static int lily;
int rose;
int stem()
{
    int rose;
    ...;
}
void root()
{
    ...;
}
```

8．下面程序会打印什么？

```
#include <stdio.h>
```

```
char color = 'B';
void first(void);
void second(void);

int main(void)
{
    extern char color;

    printf("color in main() is %c\n", color);
    first();
    printf("color in main() is %c\n", color);
    second();
    printf("color in main() is %c\n", color);
    return 0;
}

void first(void)
{
    char color;

    color = 'R';
    printf("color in first() is %c\n", color);
}

void second(void)
{
    color = 'G';
    printf("color in second() is %c\n", color);
}
```

9. 假设文件的开始处有如下声明：

```
static int plink;
int value_ct(const int arr[], int value, int n);
```

a. 以上声明表明了程序员的什么意图？

b. 用 `const int value` 和 `const int n` 分别替换 `int value` 和 `int n`，是否对主调程序的值加强保护。

12.9 编程练习

1. 不使用全局变量，重写程序清单 12.4。

2. 在美国，通常以英里/加仑来计算油耗；在欧洲，以升/100 公里来计算。下面是程序的一部分，提示用户选择计算模式（美制或公制），然后接收数据并计算油耗。

```
// pe12-2b.c
// 与 pe12-2a.c 一起编译
#include <stdio.h>
#include "pe12-2a.h"
int main(void)
{
    int mode;

    printf("Enter 0 for metric mode, 1 for US mode: ");
    scanf("%d", &mode);
    while (mode >= 0)
```

```
    {
        set_mode(mode);
        get_info();
        show_info();
        printf("Enter 0 for metric mode, 1 for US mode");
        printf(" (-1 to quit): ");
        scanf("%d", &mode);
    }
    printf("Done.\n");
    return 0;
}
```

下面是是一些输出示例：

```
Enter 0 for metric mode, 1 for US mode: 0
Enter distance traveled in kilometers: 600
Enter fuel consumed in liters: 78.8
Fuel consumption is 13.13 liters per 100 km.
Enter 0 for metric mode, 1 for US mode (-1 to quit): 1
Enter distance traveled in miles: 434
Enter fuel consumed in gallons: 12.7
Fuel consumption is 34.2 miles per gallon.
Enter 0 for metric mode, 1 for US mode (-1 to quit): 3
Invalid mode specified. Mode 1(US) used.
Enter distance traveled in miles: 388
Enter fuel consumed in gallons: 15.3
Fuel consumption is 25.4 miles per gallon.
Enter 0 for metric mode, 1 for US mode (-1 to quit): -1
Done.
```

如果用户输入了不正确的模式，程序向用户给出提示消息并使用上一次输入的正确模式。请提供 pe12-2a.h 头文件和 pe12-2a.c 源文件。源代码文件应定义 3 个具有文件作用域、内部链接的变量。一个表示模式、一个表示距离、一个表示消耗的燃料。get_info()函数根据用户输入的模式提示用户输入相应数据，并将其储存到文件作用域变量中。show_info()函数根据设置的模式计算并显示油耗。可以假设用户输入的都是数值数据。

3. 重新设计编程练习 2，要求只使用自动变量。该程序提供的用户界面不变，即提示用户输入模式等。但是，函数调用要作相应变化。

4. 在一个循环中编写并测试一个函数，该函数返回它被调用的次数。

5. 编写一个程序，生成 100 个 1～10 范围内的随机数，并以降序排列（可以把第 11 章的排序算法稍加改动，便可用于整数排序，这里仅对整数排序）。

6. 编写一个程序，生成 1000 个 1～10 范围内的随机数。不用保存或打印这些数字，仅打印每个数出现的次数。用 10 个不同的种子值运行，生成的数字出现的次数是否相同？可以使用本章自定义的函数或 ANSI C 的 rand()和 srand()函数，它们的格式相同。这是一个测试特定随机数生成器随机性的方法。

7. 编写一个程序，按照程序清单 12.13 输出示例后面讨论的内容，修改该程序。使其输出类似：

```
Enter the number of sets; enter q to stop : 18
How many sides and how many dice? 6 3
Here are 18 sets of 3 6-sided throws.
  12 10 6 9 8 14 8 15 9 14 12 17 11 7 10
  13 8 14
How many sets? Enter q to stop: q
```

8．下面是程序的一部分：

```
// pe12-8.c
#include <stdio.h>
int * make_array(int elem, int val);
void show_array(const int ar [], int n);
int main(void)
{
    int * pa;
    int size;
    int value;

    printf("Enter the number of elements: ");
    while (scanf("%d", &size) == 1 && size > 0)
    {
        printf("Enter the initialization value: ");
        scanf("%d", &value);
        pa = make_array(size, value);
        if (pa)
        {
            show_array(pa, size);
            free(pa);
        }
        printf("Enter the number of elements (<1 to quit): ");
    }
    printf("Done.\n");
    return 0;
}
```

提供make_array()和show_array()函数的定义，完成该程序。make_array()函数接受两个参数，第 1 个参数是 int 类型数组的元素个数，第 2 个参数是要赋给每个元素的值。该函数调用malloc()创建一个大小合适的数组，将其每个元素设置为指定的值，并返回一个指向该数组的指针。show_array()函数显示数组的内容，一行显示 8 个数。

9．编写一个符合以下描述的函数。首先，询问用户需要输入多少个单词。然后，接收用户输入的单词，并显示出来，使用 malloc()并回答第 1 个问题（即要输入多少个单词），创建一个动态数组，该数组内含相应的指向 char 的指针（注意，由于数组的每个元素都是指向 char 的指针，所以用于储存 malloc()返回值的指针应该是一个指向指针的指针，且它所指向的指针指向 char）。在读取字符串时，该程序应该把单词读入一个临时的 char 数组，使用 malloc()分配足够的存储空间来储存单词，并把地址存入该指针数组（该数组中每个元素都是指向 char 的指针）。然后，从临时数组中把单词拷贝到动态分配的存储空间中。因此，有一个字符指针数组，每个指针都指向一个对象，该对象的大小正好能容纳被储存的特定单词。下面是该程序的一个运行示例：

```
How many words do you wish to enter? 5
Enter 5 words now:
I enjoyed doing this exerise
Here are your words:
I
enjoyed
doing
this
exercise
```

第 13 章

文件输入/输出

本章介绍以下内容：

- 函数：fopen()、getc()、putc()、exit()、fclose()
  fprintf()、fscanf()、fgets()、fputs()
  rewind()、fseek()、ftell()、fflush()
  fgetpos()、fsetpos()、feof()、ferror()
  ungetc()、setvbuf()、fread()、fwrite()
- 如何使用 C 标准 I/O 系列的函数处理文件
- 文件模式和二进制模式、文本和二进制格式、缓冲和无缓冲 I/O
- 使用既可以顺序访问文件也可以随机访问文件的函数

文件是当今计算机系统不可或缺的部分。文件用于储存程序、文档、数据、书信、表格、图形、照片、视频和许多其他种类的信息。作为程序员，必须会编写创建文件和从文件读写数据的程序。本章将介绍相关的内容。

13.1 与文件进行通信

有时，需要程序从文件中读取信息或把信息写入文件。这种程序与文件交互的形式就是文件重定向（第 8 章介绍过）。这种方法很简单，但是有一定限制。例如，假设要编写一个交互程序，询问用户书名并把完整的书名列表保存在文件中。如果使用重定向，应该类似于：

```
books > bklist
```

用户的输入被重定向到 `bklist` 中。这样做不仅会把不符合要求的文本写入 `bklist`，而且用户也看不到要回答什么问题。

C 提供了更强大的文件通信方法，可以在程序中打开文件，然后使用特殊的 I/O 函数读取文件中的信息或把信息写入文件。在研究这些方法之前，先简要介绍一下文件的性质。

13.1.1 文件是什么

文件（*file*）通常是在磁盘或固态硬盘上的一段已命名的存储区。对我们而言，`stdio.h` 就是一个文件的名称，该文件中包含一些有用的信息。然而，对操作系统而言，文件更复杂一些。例如，大型文件会被分开储存，或者包含一些额外的数据，方便操作系统确定文件的种类。然而，这都是操作系统所关心的，程序员关心的是 C 程序如何处理文件（除非你正在编写操作系统）。

C 把文件看作是一系列连续的字节，每个字节都能被单独读取。这与 UNIX 环境中（C 的发源地）的文件结构相对应。由于其他环境中可能无法完全对应这个模型，C 提供两种文件模式：文本模式和二进制模式。

13.1.2 文本模式和二进制模式

首先，要区分文本内容和二进制内容、文本文件格式和二进制文件格式，以及文件的文本模式和二进

制模式。

所有文件的内容都以二进制形式（0 或 1）储存。但是，如果文件最初使用二进制编码的字符（例如，ASCII 或 Unicode）表示文本（就像 C 字符串那样），该文件就是文本文件，其中包含文本内容。如果文件中的二进制值代表机器语言代码或数值数据（使用相同的内部表示，假设，用于 long 或 double 类型的值）或图片或音乐编码，该文件就是二进制文件，其中包含二进制内容。

UNIX 用同一种文件格式处理文本文件和二进制文件的内容。不奇怪，鉴于 C 是作为开发 UNIX 的工具而创建的，C 和 UNIX 在文本中都使用\n（换行符）表示换行。UNIX 目录中有一个统计文件大小的计数，程序可使用该计数确定是否读到文件结尾。然而，其他系统在此之前已经有其他方法处理文件，专门用于保存文本。也就是说，其他系统已经有一种与 UNIX 模型不同的格式处理文本文件。例如，以前的 OS X Macintosh 文件用\r（回车符）表示新的一行。早期的 MS-DOS 文件用\r\n 组合表示新的一行，用嵌入的 Ctrl+Z 字符表示文件结尾，即使实际文件用添加空字符的方法使其总大小是 256 的倍数（在 Windows 中，Notepad 仍然生成 MS-DOS 格式的文本文件，但是新的编辑器可能使用类 UNIX 格式居多）。其他系统可能保持文本文件中的每一行长度相同，如有必要，用空字符填充每一行，使其长度保持一致。或者，系统可能在每行的开始标出每行的长度。

为了规范文本文件的处理，C 提供两种访问文件的途径：**二进制**模式和**文本**模式。在二进制模式中，程序可以访问文件的每个字节。而在文本模式中，程序所见的内容和文件的实际内容不同。程序以文本模式读取文件时，把本地环境表示的行末尾或文件结尾映射为 C 模式。例如，C 程序在旧式 Macintosh 中以文本模式读取文件时，把文件中的\r 转换成\n；以文本模式写入文件时，把\n 转换成\r。或者，C 文本模式程序在 MS-DOS 平台读取文件时，把\r\n 转换成\n；写入文件时，把\n 转换成\r\n。在其他环境中编写的文本模式程序也会做类似的转换。

除了以文本模式读写文本文件，还能以二进制模式读写文本文件。如果读写一个旧式 MS-DOS 文本文件，程序会看到文件中的\r 和\n 字符，不会发生映射（图 13.1 演示了一些文本）。如果要编写旧式 Mac 格式、MS-DOS 格式或 UNIX/Linux 格式的文件模式程序，应该使用二进制模式，这样程序才能确定实际的文件内容并执行相应的动作。

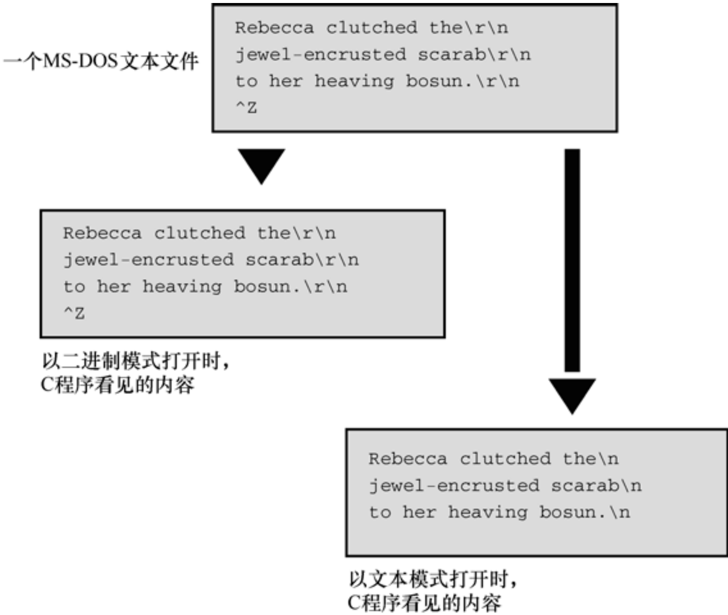


图 13.1　二进制模式和文本模式

虽然 C 提供了二进制模式和文本模式，但是这两种模式的实现可以相同。前面提到过，因为 UNIX 使用一种文件格式，这两种模式对于 UNIX 实现而言完全相同。Linux 也是如此。

13.1.3 I/O 的级别

除了选择文件的模式，大多数情况下，还可以选择 I/O 的两个级别（即处理文件访问的两个级别）。**底层** I/O（*low-level I/O*）使用操作系统提供的基本 I/O 服务。**标准高级** I/O（*standard high-level I/O*）使用 C 库的标准包和 `stdio.h` 头文件定义。因为无法保证所有的操作系统都使用相同的底层 I/O 模型，C 标准只支持标准 I/O 包。有些实现会提供底层库，但是 C 标准建立了可移植的 I/O 模型，我们主要讨论这些 I/O。

13.1.4 标准文件

C 程序会自动打开 3 个文件，它们被称为**标准输入**（*standard input*）、**标准输出**（*standard output*）和**标准错误输出**（*standard error output*）。在默认情况下，标准输入是系统的普通输入设备，通常为键盘；标准输出和标准错误输出是系统的普通输出设备，通常为显示屏。

通常，标准输入为程序提供输入，它是 `getchar()`和 `scanf()`使用的文件。程序通常输出到标准输出，它是 `putchar()`、`puts()`和 `printf()`使用的文件。第 8 章提到的重定向把其他文件视为标准输入或标准输出。标准错误输出提供了一个逻辑上不同的地方来发送错误消息。例如，如果使用重定向把输出发送给文件而不是屏幕，那么发送至标准错误输出的内容仍然会被发送到屏幕上。这样很好，因为如果把错误消息发送至文件，就只能打开文件才能看到。

13.2 标准 I/O

与底层 I/O 相比，标准 I/O 包除了可移植以外还有两个好处。第一，标准 I/O 有许多专门的函数简化了处理不同 I/O 的问题。例如，`printf()`把不同形式的数据转换成与终端相适应的字符串输出。第二，输入和输出都是**缓冲的**。也就是说，一次转移一大块信息而不是一字节信息（通常至少 512 字节）。例如，当程序读取文件时，一块数据被拷贝到缓冲区（一块中介存储区域）。这种缓冲极大地提高了数据传输速率。程序可以检查缓冲区中的字节。缓冲在后台处理，所以让人有逐字符访问的错觉（如果使用底层 I/O，要自己完成大部分工作）。程序清单 13.1 演示了如何用标准 I/O 读取文件和统计文件中的字符数。我们将在后面几节讨论程序清单 13.1 中的一些特性。该程序使用命令行参数，如果你是 Windows 用户，在编译后必须在命令提示窗口运行该程序；如果你是 Macintosh 用户，最简单的方法是使用 Terminal 在命令行形式中编译并运行该程序。或者，如第 11 章所述，如果在 IDE 中运行该程序，可以使用 Xcode 的 Product 菜单提供命令行参数。或者也可以用 `puts()`和 `fgets()`函数替换命令行参数来获得文件名。

程序清单 13.1　count.c 程序

```
/* count.c -- 使用标准 I/O */
#include <stdio.h>
#include <stdlib.h>    // 提供 exit()的原型

int main(int argc, char *argv [])
{
    int ch;             // 读取文件时，储存每个字符的地方
    FILE *fp;       // “文件指针”
    unsigned long count = 0;
    if (argc != 2)
```

```
    {
        printf("Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if ((fp = fopen(argv[1], "r")) == NULL)
    {
        printf("Can't open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
    while ((ch = getc(fp)) != EOF)
    {
        putc(ch, stdout);  // 与 putchar(ch); 相同
        count++;
    }
    fclose(fp);
    printf("File %s has %lu characters\n", argv[1], count);

    return 0;
}
```

13.2.1　检查命令行参数

首先，程序清单 13.1 中的程序检查 argc 的值，查看是否有命令行参数。如果没有，程序将打印一条消息并退出程序。字符串 argv[0]是该程序的名称。显式使用 argv[0]而不是程序名，错误消息的描述会随可执行文件名的改变而自动改变。这一特性在像 UNIX 这种允许单个文件具有多个文件名的环境中也很方便。但是，一些操作系统可能不识别 argv[0]，所以这种用法并非完全可移植。

exit()函数关闭所有打开的文件并结束程序。exit()的参数被传递给一些操作系统，包括 UNIX、Linux、Windows 和 MS-DOS，以供其他程序使用。通常的惯例是：正常结束的程序传递 0，异常结束的程序传递非零值。不同的退出值可用于区分程序失败的不同原因，这也是 UNIX 和 DOS 编程的通常做法。但是，并不是所有的操作系统都能识别相同范围内的返回值。因此，C 标准规定了一个最小的限制范围。尤其是，标准要求 0 或宏 EXIT_SUCCESS 用于表明成功结束程序，宏 EXIT_FAILURE 用于表明结束程序失败。这些宏和 exit()原型都位于 stdlib.h 头文件中。

根据 ANSI C 的规定，在最初调用的 main()中使用 return 与调用 exit()的效果相同。因此，在 main()，下面的语句：

```
return 0;
```

和下面这条语句的作用相同：

```
exit(0);
```

但是要注意，我们说的是“最初的调用”。如果 main()在一个递归程序中，exit()仍然会终止程序，但是 return 只会把控制权交给上一级递归，直至最初的一级。然后 return 结束程序。return 和 exit()的另一个区别是，即使在其他函数中（除 main()以外）调用 exit()也能结束整个程序。

13.2.2　fopen()函数

继续分析程序清单 13.1，该程序使用 fopen()函数打开文件。该函数声明在 stdio.h 中。它的第 1 个参数是待打开文件的名称，更确切地说是一个包含改文件名的字符串地址。第 2 个参数是一个字符串，指定待打开文件的模式。表 13.1 列出了 C 库提供的一些模式。

表 13.1 fopen()的模式字符串

模式字符串	含义
"r"	以读模式打开文件
"w"	以写模式打开文件，把现有文件的长度截为 0，如果文件不存在，则创建一个新文件
"a"	以写模式打开文件，在现有文件末尾添加内容，如果文件不存在，则创建一个新文件
"r+"	以更新模式打开文件（即可以读写文件）
"w+"	以更新模式打开文件（即，读和写），如果文件存在,则将其长度截为 0；如果文件不存在,则创建一个新文件
"a+"	以更新模式打开文件（即，读和写），在现有文件的末尾添加内容，如果文件不存在则创建一个新文件；可以读整个文件，但是只能从末尾添加内容
"rb"、"wb"、"ab"、"ab+"、"a+b"、"wb+"、"w+b"、"ab+"、"a+b"	与上一个模式类似，但是以二进制模式而不是文本模式打开文件
"wx"、"wbx"、"w+x"、"wb+x"或"w+bx"	（C11）类似非 x 模式，但是如果文件已存在或以独占模式打开文件，则打开文件失败

像 UNIX 和 Linux 这样只有一种文件类型的系统，带 b 字母的模式和不带 b 字母的模式相同。

新的 C11 新增了带 x 字母的写模式，与以前的写模式相比具有更多特性。第一，如果以传统的一种写模式打开一个现有文件，fopen()会把该文件的长度截为 0，这样就丢失了该文件的内容。但是使用带 x 字母的写模式，即使 fopen()操作失败，原文件的内容也不会被删除。第二，如果环境允许，x 模式的独占特性使得其他程序或线程无法访问正在被打开的文件。

警告

如果使用任何一种"w"模式（不带 x 字母）打开一个现有文件，该文件的内容会被删除，以便程序在一个空白文件中开始操作。然而，如果使用带 x 字母的任何一种模式，将无法打开一个现有文件。

程序成功打开文件后，fopen()将返回**文件指针**（*file pointer*），其他 I/O 函数可以使用这个指针指定该文件。文件指针（该例中是 fp）的类型是指向 FILE 的指针，FILE 是一个定义在 stdio.h 中的派生类型。文件指针 fp 并不指向实际的文件，它指向一个包含文件信息的数据对象，其中包含操作文件的 I/O 函数所用的缓冲区信息。因为标准库中的 I/O 函数使用缓冲区，所以它们不仅要知道缓冲区的位置，还要知道缓冲区被填充的程度以及操作哪一个文件。标准 I/O 函数根据这些信息在必要时决定再次填充或清空缓冲区。fp 指向的数据对象包含了这些信息（该数据对象是一个 C 结构，将在第 14 章中介绍）。

13.2.3 getc()和 putc()函数

getc()和 putc()函数与 getchar()和 putchar()函数类似。所不同的是，要告诉 getc()和 putc()函数使用哪一个文件。下面这条语句的意思是“从标准输入中获取一个字符”：

```
ch = getchar();
```

然而，下面这条语句的意思是“从 fp 指定的文件中获取一个字符”：

```
ch = getc(fp);
```

与此类似，下面语句的意思是“把字符 ch 放入 FILE 指针 fpout 指定的文件中”：

```
putc(ch, fpout);
```

在 putc()函数的参数列表中，第 1 个参数是待写入的字符，第 2 个参数是文件指针。

程序清单 13.1 把 stdout 作为 putc()的第 2 个参数。stdout 作为与标准输出相关联的文件指针，定义在 stdio.h 中，所以 putc(ch, stdout)与 putchar(ch)的作用相同。实际上，putchar()函数一般通过 putc()来定义。与此类似，getchar()也通过使用标准输入的 getc()来定义。

为何该示例不用 putchar()而要用 putc()？原因之一是为了介绍 putc()函数；原因之二是，把 stdout 替换成别的参数，很容易将这段程序改写成文件输出。

13.2.4　文件结尾

从文件中读取数据的程序在读到文件结尾时要停止。如何告诉程序已经读到文件结尾？如果 getc()函数在读取一个字符时发现是文件结尾，它将返回一个特殊值 EOF。所以 C 程序只有在读到超过文件末尾时才会发现文件的结尾（一些其他语言用一个特殊的函数在读取之前测试文件结尾，C 语言不同）。

为了避免读到空文件，应该使用入口条件循环（不是 do while 循环）进行文件输入。鉴于 getc()（和其他 C 输入函数）的设计，程序应该在进入循环体之前先尝试读取。如下面设计所示：

```
// 设计范例 #1
int ch;             // 用 int 类型的变量储存 EOF
FILE * fp;
fp = fopen("wacky.txt", "r");
ch = getc(fp);          // 获取初始输入
while (ch != EOF)
{
    putchar(ch);  // 处理输入
    ch = getc(fp);     // 获取下一个输入
}
```

以上代码可简化为：

```
// 设计范例 #2
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (( ch = getc(fp)) != EOF)
{
    putchar(ch);  //处理输入
}
```

由于 ch = getc(fp)是 while 测试条件的一部分，所以程序在进入循环体之前就读取了文件。不要设计成下面这样：

```
// 糟糕的设计（存在两个问题）
int ch;
FILE * fp;
fp = fopen("wacky.txt", "r");
while (ch != EOF) // 首次使用 ch 时，它的值尚未确定
{
    ch = getc(fp);     // 获取输入
    putchar(ch);       // 处理输入
}
```

第 1 个问题是，ch 首次与 EOF 比较时，其值尚未确定。第 2 个问题是，如果 getc()返回 EOF，该循

环会把 EOF 作为一个有效字符处理。这些问题都可以解决。例如，把 ch 初始化为一个哑值（*dummy value*），再把一个 if 语句加入到循环中。但是，何必多此一举，直接使用上面的设计范例即可。

其他输入函数也会用到这种处理方案，它们在读到文件结尾时也会返回一个错误信号（EOF 或 NULL 指针）。

13.2.5　fclose()函数

fclose(fp)函数关闭 fp 指定的文件，必要时刷新缓冲区。对于较正式的程序，应该检查是否成功关闭文件。如果成功关闭，fclose()函数返回 0，否则返回 EOF：

```
if (fclose(fp) != 0)
    printf("Error in closing file %s\n", argv[1]);
```

如果磁盘已满、移动硬盘被移除或出现 I/O 错误，都会导致调用 fclose()函数失败。

13.2.6　指向标准文件的指针

stdio.h 头文件把 3 个文件指针与 3 个标准文件相关联，C 程序会自动打开这 3 个标准文件。如表 13.2 所示：

表 13.2　标准文件和相关联的文件指针

标准文件	文件指针	通常使用的设备
标准输入	stdin	键盘
标准输出	stdout	显示器
标准错误	stderr	显示器

这些文件指针都是指向 FILE 的指针，所以它们可用作标准 I/O 函数的参数，如 fclose(fp)中的 fp。接下来，我们用一个程序示例创建一个新文件，并写入内容。

13.3　一个简单的文件压缩程序

下面的程序示例把一个文件中选定的数据拷贝到另一个文件中。该程序同时打开了两个文件，以"r"模式打开一个，以"w"模式打开另一个。该程序（程序清单 13.2）以保留每 3 个字符中的第 1 个字符的方式压缩第 1 个文件的内容。最后，把压缩后的文本存入第 2 个文件。第 2 个文件的名称是第 1 个文件名加上.red 后缀（此处的 red 代表 reduced）。使用命令行参数，同时打开多个文件，以及在原文件名后面加上后缀，都是相当有用的技巧。这种压缩方式有限，但是也有它的用途（很容易把该程序改成用标准 I/O 而不是命令行参数提供文件名）。

程序清单 13.2　reducto.c 程序

```
// reducto.c –把文件压缩成原来的1/3!
#include <stdio.h>
#include <stdlib.h>    // 提供 exit()的原型
#include <string.h>    // 提供 strcpy()、strcat()的原型
#define LEN 40

int main(int argc, char *argv [])
{
    FILE  *in, *out;   // 声明两个指向 FILE 的指针
```

```
    int ch;
    char name[LEN];   // 储存输出文件名
    int count = 0;

    // 检查命令行参数
    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    // 设置输入
    if ((in = fopen(argv[1], "r")) == NULL)
    {
        fprintf(stderr, "I couldn't open the file \"%s\"\n",
                argv[1]);
        exit(EXIT_FAILURE);
    }
    // 设置输出
    strncpy(name, argv[1], LEN - 5);    // 拷贝文件名
    name[LEN - 5] = '\0';
    strcat(name, ".red");               // 在文件名后添加.red
    if ((out = fopen(name, "w")) == NULL)
    {                    // 以写模式打开文件
        fprintf(stderr, "Can't create output file.\n");
        exit(3);
    }
    // 拷贝数据
    while ((ch = getc(in)) != EOF)
        if (count++ % 3 == 0)
            putc(ch, out);// 打印 3 个字符中的第 1 个字符
    // 收尾工作
    if (fclose(in) != 0 || fclose(out) != 0)
        fprintf(stderr, "Error in closing files\n");

    return 0;
}
```

假设可执行文件名是 reducto，待读取的文件名为 eddy，该文件中包含下面一行内容：

```
So even Eddy came oven ready.
```

命令如下：

```
reducto eddy
```

待写入的文件名为 eddy.red。该程序把输出显示在 eddy.red 中，而不是屏幕上。打开 eddy.red，内容如下：

```
Send money
```

该程序示例演示了几个编程技巧。我们来仔细研究一下。

fprintf()和 printf()类似，但是 fprintf()的第 1 个参数必须是一个文件指针。程序中使用 stderr 指针把错误消息发送至标准错误，C 标准通常都这么做。

为了构造新的输出文件名，该程序使用 strncpy()把名称 eddy 拷贝到数组 name 中。参数 LEN-5 为.red 后缀和末尾的空字符预留了空间。如果 argv[2]字符串比 LEN-5 长，就拷贝不了空字符。出现这

种情况时，程序会添加空字符。调用 strncpy()后，name 中的第 1 个空字符在调用 strcat()函数时，被.red 的.覆盖，生成了 eddy.red。程序中还检查了是否成功打开名为 eddy.red 的文件。这个步骤在一些环境中相当重要，因为像 strange.c.red 这样的文件名可能是无效的。例如，在传统的 DOS 环境中，不能在后缀名后面添加后缀名（MS-DOS 使用的方法是用.red 替换现有后缀名，所以 strange.c 将变成 strange.red。例如，可以用 strchr()函数定位（如果有的话），然后只拷贝点前面的部分即可）。

该程序同时打开了两个文件，所以我们要声明两个 FIFL 指针。注意，程序都是单独打开和关闭每个文件。同时打开的文件数量是有限的，这个限制取决于系统和实现，范围一般是 10～20。相同的文件指针可以处理不同的文件，前提是这些文件不需要同时打开。

13.4　文件 I/O：fprintf()、fscanf()、fgets()和 fputs()

前面章节介绍的 I/O 函数都类似于文件 I/O 函数。它们的主要区别是，文件 I/O 函数要用 FILE 指针指定待处理的文件。与 getc()、putc()类似，这些函数都要求用指向 FILE 的指针（如，stdout）指定一个文件，或者使用 fopen()的返回值。

13.4.1　fprintf()和 fscanf()函数

文件 I/O 函数 fprintf()和 fscanf()函数的工作方式与 printf()和 scanf()类似，区别在于前者需要用第 1 个参数指定待处理的文件。我们在前面用过 fprintf()。程序清单 13.3 演示了这两个文件 I/O 函数和 rewind()函数的用法。

程序清单 13.3　addaword.c 程序

```
/* addaword.c -- 使用 fprintf()、fscanf() 和 rewind() */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAX 41

int main(void)
{
    FILE *fp;
    char words[MAX];

    if ((fp = fopen("wordy", "a+")) == NULL)
    {
        fprintf(stdout, "Can't open \"wordy\" file.\n");
        exit(EXIT_FAILURE);
    }

    puts("Enter words to add to the file; press the #");
    puts("key at the beginning of a line to terminate.");
    while ((fscanf(stdin, "%40s", words) == 1) && (words[0] != '#'))
        fprintf(fp, "%s\n", words);

    puts("File contents:");
    rewind(fp);        /* 返回到文件开始处 */
    while (fscanf(fp, "%s", words) == 1)
        puts(words);
    puts("Done!");
    if (fclose(fp) != 0)
```

```
        fprintf(stderr, "Error closing file\n");

    return 0;
}
```

该程序可以在文件中添加单词。使用"a+"模式，程序可以对文件进行读写操作。首次使用该程序，它将创建 wordy 文件，以便把单词存入其中。随后再使用该程序，可以在 wordy 文件后面添加单词。虽然"a+"模式只允许在文件末尾添加内容，但是该模式下可以读整个文件。rewind()函数让程序回到文件开始处，方便 while 循环打印整个文件的内容。注意，rewind()接受一个文件指针作为参数。

下面是该程序在 UNIX 环境中的一个运行示例（可执行程序已重命名为 addword）：

```
$ addaword
Enter words to add to the file; press the Enter
key at the beginning of a line to terminate.
The fabulous programmer
#
File contents:
The
fabulous
programmer
Done!
$ addaword
Enter words to add to the file; press the Enter
key at the beginning of a line to terminate.
enchanted the
large
#
File contents:
The
fabulous
programmer
enchanted
the
large
Done!
```

如你所见，fprintf()和 fscanf()的工作方式与 printf()和 scanf()类似。但是，与 putc()不同的是，fprintf()和 fscanf()函数都把 FILE 指针作为第 1 个参数，而不是最后一个参数。

13.4.2　fgets()和 fputs()函数

第 11 章时介绍过fgets()函数。它的第 1 个参数和gets()函数一样，也是表示储存输入位置的地址（char * 类型）；第 2 个参数是一个整数，表示待输入字符串的大小 [1]；最后一个参数是文件指针，指定待读取的文件。下面是一个调用该函数的例子：

```
fgets(buf, STLEN, fp);
```

这里，buf 是 char 类型数组的名称，STLEN 是字符串的大小，fp 是指向 FILE 的指针。

fgets()函数读取输入直到第 1 个换行符的后面，或读到文件结尾，或者读取 STLEN-1 个字符（以上面的 fgets()为例）。然后，fgets()在末尾添加一个空字符使之成为一个字符串。字符串的大小是其字符数加上一个空字符。如果 fgets()在读到字符上限之前已读完一整行，它会把表示行结尾的换行符放

[1] 注意，字符串大小和字符串长度不同。前者指该字符串占用多少空间，后者指该字符串的字符个数。——译者注

在空字符前面。fgets()函数在遇到 EOF 时将返回 NULL 值，可以利用这一机制检查是否到达文件结尾；如果未遇到 EOF 则之前返回传给它的地址。

fputs()函数接受两个参数：第 1 个是字符串的地址；第 2 个是文件指针。该函数根据传入地址找到的字符串写入指定的文件中。和 puts()函数不同，fputs()在打印字符串时不会在其末尾添加换行符。下面是一个调用该函数的例子：

```
fputs(buf, fp);
```

这里，buf 是字符串的地址，fp 用于指定目标文件。

由于 fgets()保留了换行符，fputs()就不会再添加换行符，它们配合得非常好。如第 11 章的程序清单 11.8 所示，即使输入行比 STLEN 长，这两个函数依然处理得很好。

13.5 随机访问：fseek()和 ftell()

有了 fseek()函数，便可把文件看作是数组，在 fopen()打开的文件中直接移动到任意字节处。我们创建一个程序（程序清单 13.4）演示 fseek()和 ftell()的用法。注意，fseek()有 3 个参数，返回 int 类型的值；ftell()函数返回一个 long 类型的值，表示文件中的当前位置。

程序清单 13.4 reverse.c 程序

```
/* reverse.c -- 倒序显示文件的内容 */
#include <stdio.h>
#include <stdlib.h>
#define CNTL_Z '\032'      /* DOS 文本文件中的文件结尾标记 */
#define SLEN 81
int main(void)
{
    char file[SLEN];
    char ch;
    FILE *fp;
    long count, last;

    puts("Enter the name of the file to be processed:");
    scanf("%80s", file);
    if ((fp = fopen(file, "rb")) == NULL)
    {                                       /* 只读模式    */
        printf("reverse can't open %s\n", file);
        exit(EXIT_FAILURE);
    }

    fseek(fp, 0L, SEEK_END);                /* 定位到文件末尾 */
    last = ftell(fp);
    for (count = 1L; count <= last; count++)
    {
        fseek(fp, -count, SEEK_END);        /* 回退        */
        ch = getc(fp);
        if (ch != CNTL_Z && ch != '\r') /* MS-DOS 文件 */
            putchar(ch);
    }
    putchar('\n');
    fclose(fp);
```

```
    return 0;
}
```

下面是对一个文件的输出：

```
Enter the name of the file to be processed:
Cluv

.C ni eno naht ylevol erom margorp a
ees reven llahs I taht kniht I
```

该程序使用二进制模式，以便处理 MS-DOS 文本和 UNIX 文件。但是，在使用其他格式文本文件的环境中可能无法正常工作。

> **注意**
>
> 如果通过命令行环境运行该程序，待处理文件要和可执行文件在同一个目录（或文件夹）中。如果在 IDE 中运行该程序，具体查找方案序因实现而异。例如，默认情况下，Microsoft Visual Studio 2012 在源代码所在的目录中查找，而 Xcode 4.6 则在可执行文件所在的目录中查找。

接下来，我们要讨论 3 个问题：`fseek()`和 `ftell()`函数的工作原理、如何使用二进制流、如何让程序可移植。

13.5.1　fseek()和 ftell()的工作原理

`fseek()`的第 1 个参数是 `FILE` 指针，指向待查找的文件，`fopen()`应该已打开该文件。

`fseek()`的第 2 个参数是**偏移量**（*offset*）。该参数表示从起始点开始要移动的距离（参见表 13.3 列出的起始点模式）。该参数必须是一个 `long` 类型的值，可以为正（前移）、负（后移）或 `0`（保持不动）。

`fseek()`的第 3 个参数是模式，该参数确定起始点。根据 ANSI 标准，在 `stdio.h` 头文件中规定了几个表示模式的**明示常量**（*manifest constant*），如表 13.3 所示。

表 13.3　文件的起始点模式

模式	偏移量的起始点
SEEK_SET	文件开始处
SEEK_CUR	当前位置
SEEK_END	文件末尾

旧的实现可能缺少这些定义，可以使用数值 `0L`、`1L`、`2L` 分别表示这 3 种模式。`L` 后缀表明其值是 `long` 类型。或者，实现可能把这些明示常量定义在别的头文件中。如果不确定，请查阅实现的使用手册或在线帮助。

下面是调用 `fseek()`函数的一些示例，`fp` 是一个文件指针：

```
fseek(fp, 0L, SEEK_SET);   // 定位至文件开始处
fseek(fp, 10L, SEEK_SET);  // 定位至文件中的第 10 个字节
fseek(fp, 2L, SEEK_CUR);   // 从文件当前位置前移 2 个字节
fseek(fp, 0L, SEEK_END);   // 定位至文件结尾
fseek(fp, -10L, SEEK_END); // 从文件结尾处回退 10 个字节
```

对于这些调用还有一些限制，我们稍后再讨论。

如果一切正常，fseek()的返回值为 0；如果出现错误（如试图移动的距离超出文件的范围），其返回值为-1。

ftell()函数的返回类型是 long，它返回的是当前的位置。ANSI C 把它定义在 stdio.h 中。在最初实现的 UNIX 中，ftell()通过返回距文件开始处的字节数来确定文件的位置。文件的第 1 个字节到文件开始处的距离是 0，以此类推。ANSI C 规定，该定义适用于以二进制模式打开的文件，以文件模式打开文件的情况不同。这也是程序清单 13.4 以二进制模式打开文件的原因。

下面，我们来分析程序清单 13.4 中的基本要素。首先，下面的语句：

```
fseek(fp, 0L, SEEK_END);
```

把当前位置设置为距文件末尾 0 字节偏移量。也就是说，该语句把当前位置设置在文件结尾。下一条语句：

```
last = ftell(fp);
```

把从文件开始处到文件结尾的字节数赋给 last。

然后是一个 for 循环：

```
for (count = 1L; count <= last; count++)
{
    fseek(fp, -count, SEEK_END); /* go backward */
    ch = getc(fp);
}
```

第 1 轮迭代，把程序定位到文件结尾的第 1 个字符（即，文件的最后一个字符）。然后，程序打印该字符。下一轮迭代把程序定位到前一个字符，并打印该字符。重复这一过程直至到达文件的第 1 个字符，并打印。

13.5.2 二进制模式和文本模式

我们设计的程序清单 13.4 在 UNIX 和 MS-DOS 环境下都可以运行。UNIX 只有一种文件格式，所以不需要进行特殊的转换。然而 MS-DOS 要格外注意。许多 MS-DOS 编辑器都用 Ctrl+Z 标记文本文件的结尾。以文本模式打开这样的文件时，C 能识别这个作为文件结尾标记的字符。但是，以二进制模式打开相同的文件时，Ctrl+Z 字符被看作是文件中的一个字符，而实际的文件结尾符在该字符的后面。文件结尾符可能紧跟在 Ctrl+Z 字符后面，或者文件中可能用空字符填充，使该文件的大小是 256 的倍数。在 DOS 环境下不会打印空字符，程序清单 13.4 中就包含了防止打印 Ctrl+Z 字符的代码。

二进制模式和文本模式的另一个不同之处是：MS-DOS 用\r\n 组合表示文本文件换行。以文本模式打开相同的文件时，C 程序把\r\n“看成”\n。但是，以二进制模式打开该文件时，程序能看见这两个字符。因此，程序清单 13.4 中还包含了不打印\r 的代码。通常，UNIX 文本文件既没有 Ctrl+Z，也没有\r，所以这部分代码不会影响大部分 UNIX 文本文件。

ftell()函数在文本模式和二进制模式中的工作方式不同。许多系统的文本文件格式与 UNIX 的模型有很大不同，导致从文件开始处统计的字节数成为一个毫无意义的值。ANSI C 规定，对于文本模式，ftell()返回的值可以作为 fseek()的第 2 个参数。对于 MS-DOS，ftell()返回的值把\r\n 当作一个字节计数。

13.5.3 可移植性

理论上，fseek()和 ftell()应该符合 UNIX 模型。但是，不同系统存在着差异，有时确实无法做到与 UNIX 模型一致。因此，ANSI 对这些函数降低了要求。下面是一些限制。

- 在二进制模式中，实现不必支持 SEEK_END 模式。因此无法保证程序清单 13.4 的可移植性。移植

性更高的方法是逐字节读取整个文件直到文件末尾。C 预处理器的条件编译指令（第 16 章介绍）提供了一种系统方法来处理这种情况。

- 在文本模式中，只有以下调用能保证其相应的行为。

函数调用	效果
`fseek(file, 0L, SEEK_SET)`	定位至文件开始处
`fseek(file, 0L, SEEK_CUR)`	保持当前位置不动
`fseek(file, 0L, SEEK_END)`	定位至文件结尾
`fseek(file,ftell-pos, SEEK_SET)`	到距文件开始处 `ftell-pos` 的位置，`ftell-pos` 是 `ftell()`的返回值

不过，许多常见的环境都支持更多的行为。

13.5.4 fgetpos()和 fsetpos()函数

`fseek()`和 `ftell()`潜在的问题是，它们都把文件大小限制在 `long` 类型能表示的范围内。也许 20 亿字节看起来相当大，但是随着存储设备的容量迅猛增长，文件也越来越大。鉴于此，ANSI C 新增了两个处理较大文件的新定位函数：`fgetpos()`和 `fsetpos()`。这两个函数不使用 `long` 类型的值表示位置，它们使用一种新类型：`fpos_t`（代表 file position type，文件定位类型）。`fpos_t` 类型不是基本类型，它根据其他类型来定义。`fpos_t` 类型的变量或数据对象可以在文件中指定一个位置，它不能是数组类型，除此之外，没有其他限制。实现可以提供一个满足特殊平台要求的类型，例如，`fpos_t` 可以实现为结构。

ANSI C 定义了如何使用 `fpos_t` 类型。`fgetpos()`函数的原型如下：

```
int fgetpos(FILE * restrict stream, fpos_t * restrict pos);
```

调用该函数时，它把 `fpos_t` 类型的值放在 `pos` 指向的位置上，该值描述了文件中的一个位置。如果成功，`fgetpos()`函数返回 `0`；如果失败，返回非 `0`。

`fsetpos()`函数的原型如下：

```
int fsetpos(FILE *stream, const fpos_t *pos);
```

调用该函数时，使用 `pos` 指向位置上的 `fpos_t` 类型值来设置文件指针指向该值指定的位置。如果成功，`fsetpos()`函数返回 `0`；如果失败，则返回非 `0`。`fpos_t` 类型的值应通过之前调用 `fgetpos()`获得。

13.6 标准 I/O 的机理

我们在前面学习了标准 I/O 包的特性，本节研究一个典型的概念模型，分析标准 I/O 的工作原理。

通常，使用标准 I/O 的第 1 步是调用 `fopen()`打开文件（前面介绍过，C 程序会自动打开 3 种标准文件）。`fopen()`函数不仅打开一个文件，还创建了一个缓冲区（在读写模式下会创建两个缓冲区）以及一个包含文件和缓冲区数据的结构。另外，`fopen()`返回一个指向该结构的指针，以便其他函数知道如何找到该结构。假设把该指针赋给一个指针变量 `fp`，我们说 `fopen()`函数“打开一个流”。如果以文本模式打开该文件，就获得一个文本流；如果以二进制模式打开该文件，就获得一个二进制流。

这个结构通常包含一个指定流中当前位置的文件位置指示器。除此之外，它还包含错误和文件结尾的指示器、一个指向缓冲区开始处的指针、一个文件标识符和一个计数（统计实际拷贝进缓冲区的字节数）。

我们主要考虑文件输入。通常，使用标准 I/O 的第 2 步是调用一个定义在 `stdio.h` 中的输入函数，如 `fscanf()`、`getc()`或 `fgets()`。一调用这些函数，文件中的数据块就被拷贝到缓冲区中。缓冲区的大小因实现而异，一般是 512 字节或是它的倍数，如 4096 或 16384（随着计算机硬盘容量越来越大，缓冲区的大小也越来越大）。最初调用函数，除了填充缓冲区外，还要设置 `fp` 所指向的结构中的值。尤其要设置流中的当前位置和拷贝进缓冲区的字节数。通常，当前位置从字节 `0` 开始。

在初始化结构和缓冲区后，输入函数按要求从缓冲区中读取数据。在它读取数据时，文件位置指示器被设置为指向刚读取字符的下一个字符。由于 `stdio.h` 系列的所有输入函数都使用相同的缓冲区，所以调用任何一个函数都将从上一次函数停止调用的位置开始。

当输入函数发现已读完缓冲区中的所有字符时，会请求把下一个缓冲大小的数据块从文件拷贝到该缓冲区中。以这种方式，输入函数可以读取文件中的所有内容，直到文件结尾。函数在读取缓冲区中的最后一个字符后，把结尾指示器设置为真。于是，下一次被调用的输入函数将返回 `EOF`。

输出函数以类似的方式把数据写入缓冲区。当缓冲区被填满时，数据将被拷贝至文件中。

13.7 其他标准 I/O 函数

ANSI 标准库的标准 I/O 系列有几十个函数。虽然在这里无法一一列举，但是我们会简要地介绍一些，让读者对它们有一个大概的了解。这里列出函数的原型，表明函数的参数和返回类型。我们要讨论的这些函数，除了 `setvbuf()`，其他函数均可在 ANSI 之前的实现中使用。参考资料 V 的“新增 C99 和 C11 的标准 ANSI C 库”中列出了全部的 ANSI C 标准 I/O 包。

13.7.1 `int ungetc(int c, FILE *fp)`函数

`int ungetc()`函数把 c 指定的字符放回输入流中。如果把一个字符放回输入流，下次调用标准输入函数时将读取该字符（见图 13.2）。例如，假设要读取下一个冒号之前的所有字符，但是不包括冒号本身，可以使用 `getchar()`或 `getc()`函数读取字符到冒号，然后使用 `ungetc()`函数把冒号放回输入流中。ANSI C 标准保证每次只会放回一个字符。如果实现允许把一行中的多个字符放回输入流，那么下一次输入函数读入的字符顺序与放回时的顺序相反。

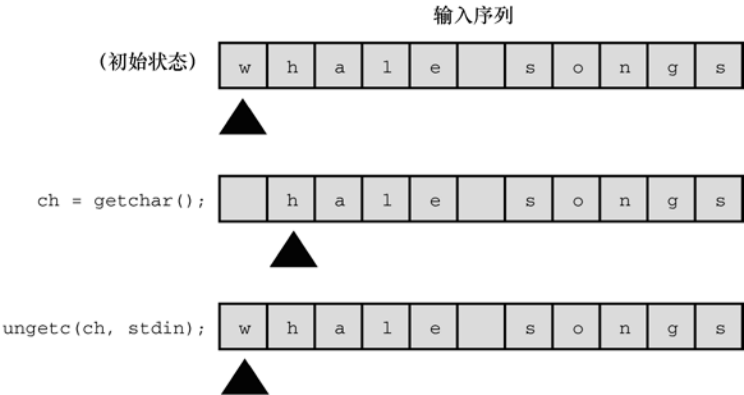


图 13.2 `ungets()`函数

13.7.2　int fflush()函数

fflush()函数的原型如下：

```
int fflush(FILE *fp);
```

调用 fflush()函数引起输出缓冲区中所有的未写入数据被发送到 fp 指定的输出文件。这个过程称为**刷新缓冲区**。如果 fp 是空指针，所有输出缓冲区都被刷新。在输入流中使用 fflush()函数的效果是未定义的。只要最近一次操作不是输入操作，就可以用该函数来更新流（任何读写模式）。

13.7.3　int setvbuf()函数

setvbuf()函数的原型是：

```
int setvbuf(FILE * restrict fp, char * restrict buf, int mode, size_t size);
```

setvbuf()函数创建了一个供标准 I/O 函数替换使用的缓冲区。在打开文件后且未对流进行其他操作之前，调用该函数。指针 fp 识别待处理的流，buf 指向待使用的存储区。如果 buf 的值不是 NULL，则必须创建一个缓冲区。例如，声明一个内含 1024 个字符的数组，并传递该数组的地址。然而，如果把 NULL 作为 buf 的值，该函数会为自己分配一个缓冲区。变量 size 告诉 setvbuf()数组的大小（size_t 是一种派生的整数类型，第 5 章介绍过）。mode 的选择如下：_IOFBF 表示完全缓冲（在缓冲区满时刷新）；_IOLBF 表示行缓冲（在缓冲区满时或写入一个换行符时）；_IONBF 表示无缓冲。如果操作成功，函数返回 0，否则返回一个非零值。

假设一个程序要储存一种数据对象，每个数据对象的大小是 3000 字节。可以使用 setvbuf()函数创建一个缓冲区，其大小是该数据对象大小的倍数。

13.7.4　二进制 I/O：fread()和 fwrite()

介绍 fread()和 fwrite()函数之前，先要了解一些背景知识。之前用到的标准 I/O 函数都是面向文本的，用于处理字符和字符串。如何要在文件中保存数值数据？用 fprintf()函数和%f 转换说明只是把数值保存为字符串。例如，下面的代码：

```
double num = 1./3.;
fprintf(fp,"%f", num);
```

把 num 储存为 8 个字符：0.333333。使用%.2f 转换说明将其储存为 4 个字符：0.33，用%.12f 转换说明则将其储存为 14 个字符：0.333333333333。改变转换说明将改变储存该值所需的空间数量，也会导致储存不同的值。把 num 储存为 0.33 后，读取文件时就无法将其恢复为更高的精度。一般而言，fprintf()把数值转换为字符数据，这种转换可能会改变值。

为保证数值在储存前后一致，最精确的做法是使用与计算机相同的位组合来储存。因此，double 类型的值应该储存在一个 double 大小的单元中。如果以程序所用的表示法把数据储存在文件中，则称以**二进制形式**储存数据。不存在从数值形式到字符串的转换过程。对于标准 I/O，fread()和 fwrite 函数用于以二进制形式处理数据（见图 13.3）。

实际上，所有的数据都是以二进制形式储存的，甚至连字符都以字符码的二进制表示来储存。如果文件中的所有数据都被解释成字符码，则称该文件包含文本数据。如果部分或所有的数据都被解释成二进制形式的数值数据，则称该文件包含二进制数据（另外，用数据表示机器语言指令的文件都是二进制文件）。

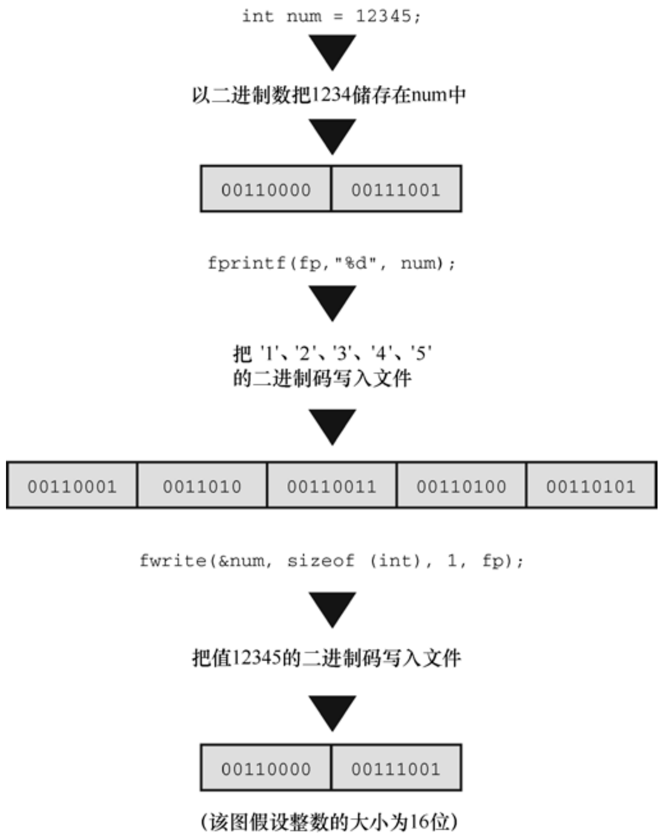


图 13.3 二进制输出和文本输出

二进制和**文本**的用法很容易混淆。ANSI C 和许多操作系统都识别两种文件格式：二进制和文本。能以二进制数据或文本数据形式存储或读取信息。可以用二进制模式打开文本格式的文件，可以把文本储存在二进制形式的文件中。可以调用 getc()拷贝包含二进制数据的文件。然而，一般而言，用二进制模式在二进制格式文件中储存二进制数据。类似地，最常用的还是以文本格式打开文本文件中的文本数据（通常文字处理器生成的文件都是二进制文件，因为这些文件中包含了大量非文本信息，如字体和格式等）。

13.7.5 size_t fwrite()函数

fwrite()函数的原型如下：

```
size_t fwrite(const void * restrict ptr, size_t size, size_t nmemb,FILE * restrict fp);
```

fwrite()函数把二进制数据写入文件。size_t 是根据标准 C 类型定义的类型，它是 sizeof 运算符返回的类型，通常是 unsigned int，但是实现可以选择使用其他类型。指针 ptr 是待写入数据块的地址。size 表示待写入数据块的大小（以字节为单位），nmemb 表示待写入数据块的数量。和其他函数一样，fp 指定待写入的文件。例如，要保存一个大小为 256 字节的数据对象（如数组），可以这样做：

```
char buffer[256];
fwrite(buffer, 256, 1, fp);
```

以上调用把一块 256 字节的数据从 buffer 写入文件。另举一例，要保存一个内含 10 个 double 类型值的数组，可以这样做：

```
double earnings[10];
fwrite(earnings, sizeof(double), 10, fp);
```

以上调用把 earnings 数组中的数据写入文件，数据被分成 10 块，每块都是 double 的大小。

注意 fwrite()原型中的 const void * restrict ptr 声明。fwrite()的一个问题是，它的第 1 个参数不是固定的类型。例如，第 1 个例子中使用 buffer，其类型是指向 char 的指针；而第 2 个例子中使用 earnings，其类型是指向 double 的指针。在 ANSI C 函数原型中，这些实际参数都被转换成指向 void 的指针类型，这种指针可作为一种通用类型指针（在 ANSI C 之前，这些参数使用 char *类型，需要把实参强制转换成 char *类型）。

fwrite()函数返回成功写入项的数量。正常情况下，该返回值就是 nmemb，但如果出现写入错误，返回值会比 nmemb 小。

13.7.6　size_t fread()函数

size_t fread()函数的原型如下：

```
size_t fread(void * restrict ptr, size_t size, size_t nmemb,FILE * restrict fp);
```

fread()函数接受的参数和 fwrite()函数相同。在 fread()函数中，ptr 是待读取文件数据在内存中的地址，fp 指定待读取的文件。该函数用于读取被 fwrite()写入文件的数据。例如，要恢复上例中保存的内含 10 个 double 类型值的数组，可以这样做：

```
double earnings[10];
fread(earnings, sizeof (double), 10, fp);
```

该调用把 10 个 double 大小的值拷贝进 earnings 数组中。

fread()函数返回成功读取项的数量。正常情况下，该返回值就是 nmemb，但如果出现读取错误或读到文件结尾，该返回值就会比 nmemb 小。

13.7.7　int feof(FILE *fp)和 int ferror(FILE *fp)函数

如果标准输入函数返回 EOF，则通常表明函数已到达文件结尾。然而，出现读取错误时，函数也会返回 EOF。feof()和 ferror()函数用于区分这两种情况。当上一次输入调用检测到文件结尾时，feof()函数返回一个非零值，否则返回 0。当读或写出现错误，ferror()函数返回一个非零值，否则返回 0。

13.7.8　一个程序示例

接下来，我们用一个程序示例说明这些函数的用法。该程序把一系列文件中的内容附加在另一个文件的末尾。该程序存在一个问题：如何给文件传递信息。可以通过交互或使用命令行参数来完成，我们先采用交互式的方法。下面列出了程序的设计方案。

- 询问目标文件的名称并打开它。
- 使用一个循环询问源文件。
- 以读模式依次打开每个源文件，并将其添加到目标文件的末尾。

为演示 setvbuf()函数的用法，该程序将使用它指定一个不同的缓冲区大小。下一步是细化程序打开目标文件的步骤：

1．以附加模式打开目标文件；

2．如果打开失败，则退出程序；

3．为该文件创建一个 4096 字节的缓冲区；

4．如果创建失败，则退出程序。

与此类似，通过以下具体步骤细化拷贝部分：

1．如果该文件与目标文件相同，则跳至下一个文件；

2．如果以读模式无法打开文件，则跳至下一个文件；

3．把文件内容添加至目标文件末尾。

最后，程序回到目标文件的开始处，显示当前整个文件的内容。

作为练习，我们使用`fread()`和`fwrite()`函数进行拷贝。程序清单 13.5 给出了这个程序。

程序清单 13.5　append.c 程序

```
/* append.c -- 把文件附加到另一个文件末尾 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define BUFSIZE 4096
#define SLEN 81
void append(FILE *source, FILE *dest);
char * s_gets(char * st, int n);

int main(void)
{
    FILE *fa, *fs;     // fa 指向目标文件，fs 指向源文件
    int files = 0;         // 附加的文件数量
    char file_app[SLEN];   // 目标文件名
    char file_src[SLEN];   // 源文件名
    int ch;

    puts("Enter name of destination file:");
    s_gets(file_app, SLEN);
    if ((fa = fopen(file_app, "a+")) == NULL)
    {
        fprintf(stderr, "Can't open %s\n", file_app);
        exit(EXIT_FAILURE);
    }
    if (setvbuf(fa, NULL, _IOFBF, BUFSIZE) != 0)
    {
        fputs("Can't create output buffer\n", stderr);
        exit(EXIT_FAILURE);
    }
    puts("Enter name of first source file (empty line to quit):");
    while (s_gets(file_src, SLEN) && file_src[0] != '\0')
    {
        if (strcmp(file_src, file_app) == 0)
            fputs("Can't append file to itself\n", stderr);
        else if ((fs = fopen(file_src, "r")) == NULL)
            fprintf(stderr, "Can't open %s\n", file_src);
        else
        {
            if (setvbuf(fs, NULL, _IOFBF, BUFSIZE) != 0)
            {
                fputs("Can't create input buffer\n", stderr);
                continue;
```

```
            }
            append(fs, fa);
            if (ferror(fs) != 0)
                fprintf(stderr, "Error in reading file %s.\n",
                        file_src);
            if (ferror(fa) != 0)
                fprintf(stderr, "Error in writing file %s.\n",
                        file_app);
            fclose(fs);
            files++;
            printf("File %s appended.\n", file_src);
            puts("Next file (empty line to quit):");
        }
    }
    printf("Done appending. %d files appended.\n", files);
    rewind(fa);
    printf("%s contents:\n", file_app);
    while ((ch = getc(fa)) != EOF)
        putchar(ch);
    puts("Done displaying.");
    fclose(fa);

    return 0;
}

void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE]; // 只分配一次

    while ((bytes = fread(temp, sizeof(char), BUFSIZE, source)) > 0)
        fwrite(temp, sizeof(char), bytes, dest);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

如果 setvbuf()无法创建缓冲区，则返回一个非零值，然后终止程序。可以用类似的代码为正在拷贝的文件创建一块 4096 字节的缓冲区。把 NULL 作为 setvbuf()的第 2 个参数，便可让函数分配缓冲区的

存储空间。

该程序获取文件名所用的函数是 s_gets()，而不是 scanf()，因为 scanf()会跳过空白，因此无法检测到空行。该程序还用 s_gets()代替 fgets()，因为后者在字符串中保留换行符。

以下代码防止程序把文件附加在自身末尾：

```
if (strcmp(file_src, file_app) == 0)
    fputs("Can't append file to itself\n",stderr);
```

参数 file_app 表示目标文件名，file_src 表示正在处理的文件名。

append()函数完成拷贝任务。该函数使用 fread()和 fwrite()一次拷贝 4096 字节，而不是一次拷贝 1 字节：

```
void append(FILE *source, FILE *dest)
{
    size_t bytes;
    static char temp[BUFSIZE]; // 只分配一次
    while ((bytes = fread(temp, sizeof(char), BUFSIZE, source)) > 0)
        fwrite(temp, sizeof(char), bytes, dest);
}
```

因为是以附加模式打开由 dest 指定的文件，所以所有的源文件都被依次添加至目标文件的末尾。注意，temp 数组具有静态存储期（意思是在编译时分配该数组，不是在每次调用 append()函数时分配）和块作用域（意思是该数组属于它所在的函数私有）。

该程序示例使用文本模式的文件。使用"ab+"和"rb"模式可以处理二进制文件。

13.7.9 用二进制 I/O 进行随机访问

随机访问是用二进制 I/O 写入二进制文件最常用的方式，我们来看一个简短的例子。程序清单 13.6 中的程序创建了一个储存 double 类型数字的文件，然后让用户访问这些内容。

程序清单 13.6 randbin.c 程序

```
/* randbin.c -- 用二进制 I/O 进行随机访问 */
#include <stdio.h>
#include <stdlib.h>
#define ARSIZE 1000

int main()
{
    double numbers[ARSIZE];
    double value;
    const char * file = "numbers.dat";
    int i;
    long pos;
    FILE *iofile;

    // 创建一组 double 类型的值
    for (i = 0; i < ARSIZE; i++)
        numbers[i] = 100.0 * i + 1.0 / (i + 1);
    // 尝试打开文件
    if ((iofile = fopen(file, "wb")) == NULL)
    {
        fprintf(stderr, "Could not open %s for output.\n", file);
        exit(EXIT_FAILURE);
```

```
    }
    // 以二进制格式把数组写入文件
    fwrite(numbers, sizeof(double), ARSIZE, iofile);
    fclose(iofile);
    if ((iofile = fopen(file, "rb")) == NULL)
    {
        fprintf(stderr,
            "Could not open %s for random access.\n", file);
        exit(EXIT_FAILURE);
    }
    // 从文件中读取选定的内容
    printf("Enter an index in the range 0-%d.\n", ARSIZE - 1);
    while (scanf("%d", &i) == 1 && i >= 0 && i < ARSIZE)
    {
        pos = (long) i * sizeof(double);    // 计算偏移量
        fseek(iofile, pos, SEEK_SET);       // 定位到此处
        fread(&value, sizeof(double), 1, iofile);
        printf("The value there is %f.\n", value);
        printf("Next index (out of range to quit):\n");
    }
    // 完成
    fclose(iofile);
    puts("Bye!");

    return 0;
}
```

首先，该程序创建了一个数组，并在该数组中存放了一些值。然后，程序以二进制模式创建了一个名为 `numbers.dat` 的文件，并使用 `fwrite()`把数组中的内容拷贝到文件中。内存中数组的所有 `double` 类型值的位组合（每个位组合都是 64 位）都被拷贝至文件中。不能用文本编辑器读取最后的二进制文件，因为无法把文件中的值转换成字符串。然而，储存在文件中的每个值都与储存在内存中的值完全相同，没有损失任何精确度。此外，每个值在文件中也同样占用 64 位存储空间，所以可以很容易地计算出每个值的位置。

程序的第 2 部分用于打开待读取的文件，提示用户输入一个值的索引。程序通过把索引值和 `double` 类型值占用的字节相乘，即可得出文件中的一个位置。然后，程序调用 `fseek()`定位到该位置，用 `fread()` 读取该位置上的数据值。注意，这里并未使用转换说明。`fread()`从已定位的位置开始，拷贝 8 字节到内存中地址为`&value` 的位置。然后，使用 `printf()`显示 `value`。下面是该程序的一个运行示例：

```
Enter an index in the range 0-999.
500
The value there is 50000.001996.
Next index (out of range to quit):
900
The value there is 90000.001110.
Next index (out of range to quit):
0
The value there is 1.000000.
Next index (out of range to quit):
-1
Bye!
```

13.8　关键概念

C 程序把输入看作是字节流，输入流来源于文件、输入设备（如键盘），或者甚至是另一个程序的输出。类似地，C 程序把输出也看作是字节流，输出流的目的地可以是文件、视频显示等。

C 如何解释输入流或输出流取决于所使用的输入/输出函数。程序可以不做任何改动地读取和存储字节，或者把字节依次解释成字符，随后可以把这些字符解释成普通文本以用文本表示数字。类似地，对于输出，所使用的函数决定了二进制值是被原样转移，还是被转换成文本或以文本表示数字。如果要在不损失精度的前提下保存或恢复数值数据，请使用二进制模式以及 `fread()`和 `fwrite()`函数。如果打算保存文本信息并创建能在普通文本编辑器查看的文本，请使用文本模式和函数（如 `getc()`和 `fprintf()`）。

要访问文件，必须创建文件指针（类型是 `FILE *`）并把指针与特定文件名相关联。随后的代码就可以使用这个指针（而不是文件名）来处理该文件。

要重点理解 C 如何处理文件结尾。通常，用于读取文件的程序使用一个循环读取输入，直至到达文件结尾。C 输入函数在读过文件结尾后才会检测到文件结尾，这意味着应该在尝试读取之后立即判断是否是文件结尾。可以使用 13.2.4 节中“设计范例”中的双文件输入模式。

13.9　本章小结

对于大多数 C 程序而言，写入文件和读取文件必不可少。为此，绝大对数 C 实现都提供底层 I/O 和标准高级 I/O。因为 ANSI C 库考虑到可移植性，包含了标准 I/O 包，但是未提供底层 I/O。

标准 I/O 包自动创建输入和输出缓冲区以加快数据传输。`fopen()`函数为标准 I/O 打开一个文件，并创建一个用于存储文件和缓冲区信息的结构。`fopen()`函数返回指向该结构的指针，其他函数可以使用该指针指定待处理的文件。`feof()`和 `ferror()`函数报告 I/O 操作失败的原因。

C 把输入视为字节流。如果使用 `fread()`函数，C 把输入看作是二进制值并将其储存在指定存储位置。如果使用 `fscanf()`、`getc()`、`fgets()`或其他相关函数，C 则将每个字节看作是字符码。然后 `fscanf()`和 `scanf()`函数尝试把字符码翻译成转换说明指定的其他类型。例如，输入一个值 23，`%f` 转换说明会把 23 翻译成一个浮点值，`%d` 转换说明会把 23 翻译成一个整数值，`%s` 转换说明则会把 23 储存为字符串。`getc()`和 `fgetc()`系列函数把输入作为字符码储存，将其作为单独的字符保存在字符变量中或作为字符串储存在字符数组中。类似地，`fwrite()`将二进制数据直接放入输出流，而其他输出函数把非字符数据转换成用字符表示后才将其放入输出流。

ANSI C 提供两种文件打开模式：二进制和文本。以二进制模式打开文件时，可以逐字节读取文件；以文本模式打开文件时，会把文件内容从文本的系统表示法映射为 C 表示法。对于 UNIX 和 Linux 系统，这两种模式完全相同。

通常，输入函数 `getc()`、`fgets()`、`fscanf()`和 `fread()`都从文件开始处按顺序读取文件。然而，`fseek()`和 `ftell()`函数让程序可以随机访问文件中的任意位置。`fgetpos()`和 `fsetpos()`把类似的功能扩展至更大的文件。与文本模式相比，二进制模式更容易进行随机访问。

13.10　复习题

复习题的参考答案在附录 A 中。

1. 下面的程序有什么问题？

```
int main(void)
{
```

```
    int * fp;
    int k;

    fp = fopen("gelatin");
    for (k = 0; k < 30; k++)
        fputs(fp, "Nanette eats gelatin.");
    fclose("gelatin");
    return 0;
}
```

2. 下面的程序完成什么任务？（假设在命令行环境中运行）

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main(int argc, char *argv [])
{
    int ch;
    FILE *fp;
    if (argc < 2)
        exit(EXIT_FAILURE);
    if ((fp = fopen(argv[1], "r")) == NULL)
        exit(EXIT_FAILURE);
    while ((ch = getc(fp)) != EOF)
        if (isdigit(ch))
            putchar(ch);
    fclose(fp);

    return 0;
}
```

3. 假设程序中有下列语句：

```
#include <stdio.h>
FILE * fp1,* fp2;
char ch;

fp1 = fopen("terky", "r");
fp2 = fopen("jerky", "w");
```

另外，假设成功打开了两个文件。补全下面函数调用中缺少的参数：

a. `ch = getc();`

b. `fprintf( ,"%c\n", );`

c. `putc( , );`

d. `fclose(); /* 关闭terky文件 */`

4. 编写一个程序，不接受任何命令行参数或接受一个命令行参数。如果有一个参数，将其解释为文件名；如果没有参数，使用标准输入（stdin）作为输入。假设输入完全是浮点数。该程序要计算和报告输入数字的算术平均值。

5. 编写一个程序，接受两个命令行参数。第 1 个参数是字符，第 2 个参数是文件名。要求该程序只打印文件中包含给定字符的那些行。

> **注意**
>
> C 程序根据`'\n'`识别文件中的行。假设所有行都不超过 256 个字符，你可能会想到用 `fgets()`。

6. 二进制文件和文本文件有何区别？二进制流和文本流有何区别？
7.
   a. 分别用 `fprintf()`和 `fwrite()`储存 8238201 有何区别？
   b. 分别用 `putc()`和 `fwrite()`储存字符 S 有何区别？
8. 下面语句的区别是什么？

```
printf("Hello, %s\n", name);
fprintf(stdout, "Hello, %s\n", name);
fprintf(stderr, "Hello, %s\n", name);
```

9. `"a+"`、`"r+"`和`"w+"`模式打开的文件都是可读写的。哪种模式更适合用来更改文件中已有的内容？

13.11 编程练习

1. 修改程序清单 13.1 中的程序，要求提示用户输入文件名，并读取用户输入的信息，不使用命令行参数。
2. 编写一个文件拷贝程序，该程序通过命令行获取原始文件名和拷贝文件名。尽量使用标准 I/O 和二进制模式。
3. 编写一个文件拷贝程序，提示用户输入文本文件名，并以该文件名作为原始文件名和输出文件名。该程序要使用 `ctype.h` 中的 `toupper()`函数，在写入到输出文件时把所有文本转换成大写。使用标准 I/O 和文本模式。
4. 编写一个程序，按顺序在屏幕上显示命令行中列出的所有文件。使用 `argc` 控制循环。
5. 修改程序清单 13.5 中的程序，用命令行界面代替交互式界面。
6. 使用命令行参数的程序依赖于用户的内存如何正确地使用它们。重写程序清单 13.2 中的程序，不使用命令行参数，而是提示用户输入所需信息。
7. 编写一个程序打开两个文件。可以使用命令行参数或提示用户输入文件名。
   a. 该程序以这样的顺序打印：打印第 1 个文件的第 1 行，第 2 个文件的第 1 行，第 1 个文件的第 2 行，第 2 个文件的第 2 行，以此类推，打印到行数较多文件的最后一行。
   b. 修改该程序，把行号相同的行打印成一行。
8. 编写一个程序，以一个字符和任意文件名作为命令行参数。如果字符后面没有参数，该程序读取标准输入；否则，程序依次打开每个文件并报告每个文件中该字符出现的次数。文件名和字符本身也要一同报告。程序应包含错误检查，以确定参数数量是否正确和是否能打开文件。如果无法打开文件，程序应报告这一情况，然后继续处理下一个文件。
9. 修改程序清单 13.3 中的程序，从 1 开始，根据加入列表的顺序为每个单词编号。当程序下次运行时，确保新的单词编号接着上次的编号开始。
10. 编写一个程序打开一个文本文件，通过交互方式获得文件名。通过一个循环，提示用户输入一个文件位置。然后该程序打印从该位置开始到下一个换行符之前的内容。用户输入负数或非数值字符可以结束输入循环。
11. 编写一个程序，接受两个命令行参数。第 1 个参数是一个字符串，第 2 个参数是一个文件名。然后该程序查找该文件，打印文件中包含该字符串的所有行。因为该任务是面向行而不是面向字符的，所以要使用 `fgets()`而不是 `getc()`。使用标准 C 库函数 `strstr()`（11.5.7 节简要介绍过）在每一行中查找指定字符串。假设文件中的所有行都不超过 255 个字符。

12. 创建一个文本文件，内含 20 行，每行 30 个整数。这些整数都在 0～9 之间，用空格分开。该文件是用数字表示一张图片，0～9 表示逐渐增加的灰度。编写一个程序，把文件中的内容读入一个 20×30 的 int 数组中。一种把这些数字转换为图片的粗略方法是：该程序使用数组中的值初始化一个 20×31 的字符数组，用值 0 对应空格字符，1 对应点字符，以此类推。数字越大表示字符所占的空间越大。例如，用#表示 9。每行的最后一个字符（第 31 个）是空字符，这样该数组包含了 20 个字符串。最后，程序显示最终的图片（即，打印所有的字符串），并将结果储存在文本文件中。例如，下面是开始的数据：

```
0 0 9 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 2 0 0 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 0 5 8 9 9 8 5 5 2 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 1 9 8 5 4 5 2 0 0 0 0 0 0 0 0
0 0 0 0 9 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 4 5 2 0 0 0 0 0 0 0
0 0 9 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 4 5 2 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 1 8 5 0 0 0 4 5 2 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 4 5 2 0 0 0 0
5 5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5
8 8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8
9 9 9 0 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 3 9 9 9 9 9 9 9
8 8 8 8 8 8 8 8 8 8 8 8 5 8 9 9 8 5 8 8 8 8 8 8 8 8 8 8 8 8
5 5 5 5 5 5 5 5 5 5 5 5 5 5 8 9 9 8 5 5 5 5 5 5 5 5 5 5 5 5
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 6 6 0 0 0 0 0
0 0 0 0 2 2 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0
0 0 0 0 3 3 0 0 0 0 0 0 0 5 8 9 9 8 5 0 5 6 1 1 1 1 6 5 0 0
0 0 0 0 4 4 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 5 6 0 0 6 5 0 0 0
0 0 0 0 5 5 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 6 6 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 5 8 9 9 8 5 0 0 0 0 0 0 0 0 0 0 0
```

根据以上描述选择特定的输出字符，最终输出如下：

```
  #          *%##%*'
    #        *%##%**'
             *%.#%*~*'
    #        *%##%* ~*'
  #          *%##%*  ~*'
             *%#.%*   ~*'
             *%##%*    ~*'
*************%##%*************
%%%%%%%%%%%%*%##%*%%%%%%%%%%%%
#### ################:#######
%%%%%%%%%%%%*%##%*%%%%%%%%%%%%
*************%##%*************
             *%##%*
             *%##%*    ==
    ''       *%##%*  *=  =*
    ::       *%##%* *=....=*
    ~~       *%##%*  *=  =*
    **       *%##%*    ==
             *%##%*
             *%##%*
```

13. 用变长数组（VLA）代替标准数组，完成编程练习 12。

14. 数字图像，尤其是从宇宙飞船发回的数字图像，可能会包含一些失真。为编程练习 12 添加消除失真的函数。该函数把每个值与它上下左右相邻的值作比较，如果该值与其周围相邻值的差都大于 1，则用所有相邻值的平均值（四舍五入为整数）代替该值。注意，与边界上的点相邻的点少于 4 个，所以做特殊处理。

第 14 章

结构和其他数据形式

本章介绍以下内容：

- 关键字：struct、union、typedef
- 运算符：.、->
- 什么是 C 结构，如何创建结构模板和结构变量
- 如何访问结构的成员，如何编写处理结构的函数
- 联合和指向函数的指针

设计程序时，最重要的步骤之一是选择表示数据的方法。在许多情况下，简单变量甚至是数组还不够。为此，C 提供了**结构变量**（*structure variable*）提高你表示数据的能力，它能让你创造新的形式。如果熟悉 Pascal 的**记录**（*record*），应该很容易理解结构。如果不懂 Pascal 也没关系，本章将详细介绍 C 结构。我们先通过一个示例来分析为何需要 C 结构，学习如何创建和使用结构。

14.1　示例问题：创建图书目录

Gwen Glenn 要打印一份图书目录。她想打印每本书的各种信息：书名、作者、出版社、版权日期、页数、册数和价格。其中的一些项目（如，书名)可以储存在字符数组中，其他项目需要一个 `int` 数组或 `float` 数组。用 7 个不同的数组分别记录每一项比较繁琐，尤其是 Gwen 还想创建多份列表：一份按书名排序、一份按作者排序、一份按价格排序等。如果能把图书目录的信息都包含在一个数组里更好，其中每个元素包含一本书的相关信息。

因此，Gwen 需要一种即能包含字符串又能包含数字的数据形式，而且还要保持各信息的独立。C 结构就满足这种情况下的需求。我们通过一个示例演示如何创建和使用数组。但是，示例进行了一些限制。第一，该程序示例演示的书目只包含书名、作者和价格。第二，只有一本书的数目。当然，别忘了这只是进行了限制，我们在后面将扩展该程序。请看程序清单 14.1 及其输出，然后阅读后面的一些要点。

程序清单 14.1　book.c 程序

```
//* book.c -- 一本书的图书目录 */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL  41    /* 书名的最大长度 + 1    */
#define MAXAUTL  31    /* 作者姓名的最大长度 + 1*/

struct book {          /* 结构模版：标记是 book */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
```

```
};                          /* 结构模版结束          */

int main(void)
{
    struct book library;   /* 把 library 声明为一个 book 类型的变量 */

    printf("Please enter the book title.\n");
    s_gets(library.title, MAXTITL);     /* 访问 title 部分*/
    printf("Now enter the author.\n");
    s_gets(library.author, MAXAUTL);
    printf("Now enter the value.\n");
    scanf("%f", &library.value);
    printf("%s by %s: $%.2f\n", library.title,
        library.author, library.value);
    printf("%s: \"%s\" ($%.2f)\n", library.author,
          library.title, library.value);
    printf("Done.\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;          //处理输入行中剩余的字符
    }
    return ret_val;
}
```

我们使用前面章节中介绍的 `s_gets()`函数去掉 `fgets()`储存在字符串中的换行符。下面是该例的一个运行示例：

```
Please enter the book title.
Chicken of the Andes
Now enter the author.
Disma Lapoult
Now enter the value.
29.99
Chicken of the Andes by Disma Lapoult: $29.99
Disma Lapoult: "Chicken of the Andes" ($29.99)
Done.
```

程序清单 14.1 中创建的结构有 3 部分，每个部分都称为**成员**（*member*）或**字段**（*field*）。这 3 部分中，一部分储存书名，一部分储存作者名，一部分储存价格。下面是必须掌握的 3 个技巧：

- 为结构建立一个格式或样式；
- 声明一个适合该样式的变量；
- 访问结构变量的各个部分。

14.2 建立结构声明

结构声明（*structure declaration*）描述了一个结构的组织布局。声明类似下面这样：

```
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

该声明描述了一个由两个字符数组和一个 `float` 类型变量组成的结构。该声明并未创建实际的数据对象，只描述了该对象由什么组成。〔有时，我们把结构声明称为模板，因为它勾勒出结构是如何储存数据的。如果读者知道 C++的模板，此模板非彼模板，C++中的模板更为强大。〕我们来分析一些细节。首先是关键字 `struct`，它表明跟在其后的是一个结构，后面是一个可选的**标记**（该例中是 `book`），稍后程序中可以使用该标记引用该结构。所以，我们在后面的程序中可以这样声明：

```
struct book library;
```

这把 `library` 声明为一个使用 `book` 结构布局的结构变量。

在结构声明中，用一对花括号括起来的是结构成员列表。每个成员都用自己的声明来描述。例如，`title` 部分是一个内含 `MAXTITL` 个元素的 `char` 类型数组。成员可以是任意一种 C 的数据类型，甚至可以是其他结构！右花括号后面的分号是声明所必需的，表示结构布局定义结束。可以把这个声明放在所有函数的外部（如本例所示），也可以放在一个函数定义的内部。如果把结构声明置于一个函数的内部，它的标记就只限于该函数内部使用。如果把结构声明置于函数的外部，那么该声明之后的所有函数都能使用它的标记。例如，在程序的另一个函数中，可以这样声明：

```
struct book dickens;
```

这样，该函数便创建了一个结构变量 `dickens`，该变量的结构布局是 `book`。

结构的标记名是可选的。但是以程序示例中的方式建立结构时（在一处定义结构布局，在另一处定义实际的结构变量），必须使用标记。我们学完如何定义结构变量后，再来看这一点。

14.3 定义结构变量

结构有两层含义。一层含义是“结构布局”，刚才已经讨论过了。结构布局告诉编译器如何表示数据，但是它并未让编译器为数据**分配**空间。下一步是创建一个**结构变量**，即是结构的另一层含义。程序中创建结构变量的一行是:

```
struct book library;
```

编译器执行这行代码便创建了一个结构变量 `library`。编译器使用 `book` 模板为该变量分配空间：一个内含 `MAXTITL` 个元素的 `char` 数组、一个内含 `MAXAUTL` 个元素的 `char` 数组和一个 `float` 类型的变量。这些存储空间都与一个名称 `library` 结合在一起（见图 14.1）。

在结构变量的声明中，`struct book` 所起的作用相当于一般声明中的 `int` 或 `float`。例如，可以定义两个 `struct book` 类型的变量，或者甚至是指向 `struct book` 类型结构的指针：

```
struct book doyle, panshin, * ptbook;
```

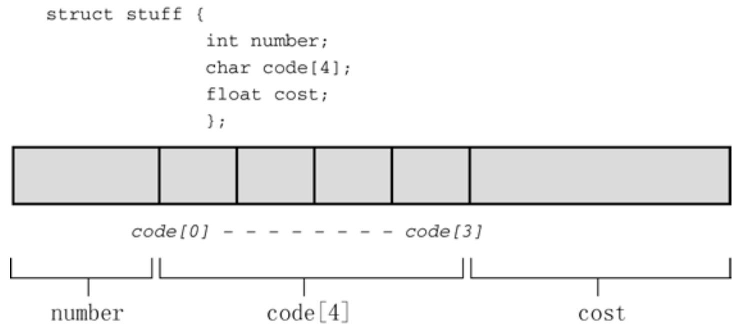


图 14.1　一个结构的内存分配

结构变量 doyle 和 panshin 中都包含 title、author 和 value 部分。指针 ptbook 可以指向 doyle、panshin 或任何其他 book 类型的结构变量。从本质上看，book 结构声明创建了一个名为 struct book 的新类型。

就计算机而言，下面的声明：

```
struct book library;
```

是以下声明的简化：

```
struct book {
    char title[MAXTITL];
    char author[AXAUTL];
    float value;
} library;    /* 声明的右右花括号后跟变量名*/
```

换言之，声明结构的过程和定义结构变量的过程可以组合成一个步骤。如下所示，组合后的结构声明和结构变量定义不需要使用结构标记：

```
struct { /* 无结构标记 */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
} library;
```

然而，如果打算多次使用结构模板，就要使用带标记的形式；或者，使用本章后面介绍的 typedef。

这是定义结构变量的一个方面，在这个例子中，并未初始化结构变量。

14.3.1　初始化结构

初始化变量和数组如下：

```
int count = 0;
int fibo[7] = {0,1,1,2,3,5,8};
```

结构变量是否也可以这样初始化？是的，可以。初始化一个结构变量（ANSI 之前，不能用自动变量初始化结构；ANSI 之后可以用任意存储类别）与初始化数组的语法类似：

```
struct book library = {
    "The Pious Pirate and the Devious Damsel",
    "Renee Vivotte",
    1.95
};
```

简而言之，我们使用在一对花括号中括起来的初始化列表进行初始化，各初始化项用逗号分隔。因此，title 成员可以被初始化为一个字符串，value 成员可以被初始化为一个数字。为了让初始化项与结构中

各成员的关联更加明显，我们让每个成员的初始化项独占一行。这样做只是为了提高代码的可读性，对编译器而言，只需要用逗号分隔各成员的初始化项即可。

> **注意 初始化结构和类别储存期**
>
> 第 12 章中提到过，如果初始化静态存储期的变量（如，静态外部链接、静态内部链接或静态无链接），必须使用常量值。这同样适用于结构。如果初始化一个静态存储期的结构，初始化列表中的值必须是常量表达式。如果是自动存储期，初始化列表中的值可以不是常量。

14.3.2 访问结构成员

结构类似于一个“超级数组”，这个超级数组中，可以是一个元素为 char 类型，下一个元素为 forat 类型，下一个元素为 int 数组。可以通过数组下标单独访问数组中的各元素，那么，如何访问结构中的成员？使用结构成员运算符——点（.）访问结构中的成员。例如，library.value 即访问 library 的 value 部分。可以像使用任何 float 类型变量那样使用 library.value。与此类似，可以像使用字符数组那样使用 library.title。因此，程序清单 14.1 中的程序中有 s_gets(library.title, MAXTITL);和 scanf("%f", &library.value);这样的代码。

本质上，.title、.author 和.value 的作用相当于 book 结构的下标。

注意，虽然 library 是一个结构，但是 library.value 是一个 float 类型的变量，可以像使用其他 float 类型变量那样使用它。例如，scanf("%f",...)需要一个 float 类型变量的地址，而&library.float 正好符合要求。.比&的优先级高，因此这个表达式和&(library.float)一样。

如果还有一个相同类型的结构变量，可以用相同的方法：

```
struct book bill, newt;

s_gets(bill.title, MAXTITL);
s_gets(newt.title, MAXTITL);
```

.title 引用 book 结构的第 1 个成员。注意，程序清单 14.1 中的程序以两种不同的格式打印了 library 结构变量中的内容。这说明可以自行决定如何使用结构成员。

14.3.3 结构的初始化器

C99 和C11 为结构提供了指定初始化器（*designated initializer*）[1]，其语法与数组的指定初始化器类似。但是，结构的指定初始化器使用点运算符和成员名（而不是方括号和下标）标识特定的元素。例如，只初始化book结构的value成员，可以这样做：

```
struct book surprise = { .value = 10.99};
```

可以按照任意顺序使用指定初始化器：

```
struct book gift = { .value = 25.99,
                     .author = "James Broadfool",
                     .title = "Rue for the Toad"};
```

与数组类似，在指定初始化器后面的普通初始化器，为指定成员后面的成员提供初始值。另外，对特定成员的最后一次赋值才是它实际获得的值。例如，考虑下面的代码：

[1] 也被称为标记化结构初始化语法。——译者注

```
struct book gift= {.value = 18.90,
                   .author = "Philionna Pestle",
                   0.25};
```

赋给 value 的值是 0.25，因为它在结构声明中紧跟在 author 成员之后。新值 0.25 取代了之前的 18.9。在学习了结构的基本知识后，可以进一步了解结构的一些相关类型。

14.4　结构数组

接下来，我们要把程序清单 14.1 的程序扩展成可以处理多本书。显然，每本书的基本信息都可以用一个 book 类型的结构变量来表示。为描述两本书，需要使用两个变量，以此类推。可以使用这一类型的结构数组来处理多本书。在下一个程序中（程序清单 14.2）就创建了一个这样的数组。如果你使用 Borland C/C++，请参阅本节后面的“Borland C 和浮点数”。

结构和内存

manybook.c 程序创建了一个内含 100 个结构变量的数组。由于该数组是自动存储类别的对象，其中的信息被储存在栈（stack）中。如此大的数组需要很大一块内存，这可能会导致一些问题。如果在运行时出现错误，可能抱怨栈大小或栈溢出，你的编译器可能使用了一个默认大小的栈，这个栈对于该例而言太小。要修正这个问题，可以使用编译器选项设置栈大小为 10000，以容纳这个结构数组；或者可以创建静态或外部数组（这样，编译器就不会把数组放在栈中）；或者可以减小数组大小为 16。为何不一开始就使用较小的数组？这是为了让读者意识到栈大小的潜在问题，以便今后再遇到类似的问题，可以自己处理好。

程序清单 14.2　manybook.c 程序

```
/* manybook.c -- 包含多本书的图书目录 */
#include <stdio.h>
#include <string.h>
char * s_gets(char * st, int n);
#define MAXTITL   40
#define MAXAUTL   40
#define MAXBKS   100        /* 书籍的最大数量  */

struct book {               /* 简历 book 模板    */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS];   /*  book 类型结构的数组 */
    int count = 0;
    int index;

    printf("Please enter the book title.\n");
    printf("Press [enter] at the start of a line to stop.\n");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
        && library[count].title[0] != '\0')
    {
        printf("Now enter the author.\n");
```

```
        s_gets(library[count].author, MAXAUTL);
        printf("Now enter the value.\n");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue;      /* 清理输入行*/
        if (count < MAXBKS)
            printf("Enter the next title.\n");
    }

    if (count > 0)
    {
        printf("Here is the list of your books:\n");
        for (index = 0; index < count; index++)
            printf("%s by %s: $%.2f\n", library[index].title,
            library[index].author, library[index].value);
    }
    else
        printf("No books? Too bad.\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // 查找换行符
        if (find)                   // 如果地址不是 NULL,
            *find = '\0';           // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;           // 处理输入行中剩余的字符
    }
    return ret_val;
}
```

下面是该程序的一个输出示例：

```
Please enter the book title.
Press [enter] at the start of a line to stop.
My Life as a Budgie
Now enter the author.
Mack Zackles
Now enter the value.
12.95
Enter the next title.
...（此处省略了许多内容）...
Here is the list of your books:
My Life as a Budgie by Mack Zackles: $12.95
Thought and Unthought Rethought by Kindra Schlagmeyer: $43.50
Concerto for Financial Instruments by Filmore Walletz: $49.99
The CEO Power Diet by Buster Downsize: $19.25
```

```
C++ Primer Plus by Stephen Prata: $59.99
Fact Avoidance: Perception as Reality by Polly Bull: $19.97
Coping with Coping by Dr. Rubin Thonkwacker: $0.02
Diaphanous Frivolity by Neda McFey: $29.99
Murder Wore a Bikini by Mickey Splats: $18.95
A History of Buvania, Volume 8, by Prince Nikoli Buvan: $50.04
Mastering Your Digital Watch, 5nd Edition, by Miklos Mysz: $28.95
A Foregone Confusion by Phalty Reasoner: $5.99
Outsourcing Government: Selection vs. Election by Ima Pundit: $33.33
```

Borland C 和浮点数

如果程序不使用浮点数，旧式的 Borland C 编译器会尝试使用小版本的 scanf()来压缩程序。然而，如果在一个结构数组中只有一个浮点值(如程序清单 14.2 中那样)，那么这种编译器(DOS 的 Borland C/C++ 3.1 之前的版本，不是 Borland C/C++ 4.0）就无法发现它存在。结果，编译器会生成如下消息:

```
scanf : floating point formats not linked
Abnormal program termination
```

一种解决方案是，在程序中添加下面的代码:

```
#include <math.h>
double dummy = sin(0.0);
```

这段代码强制编译器载入浮点版本的 scanf()。

首先，我们学习如何声明结构数组和如何访问数组中的结构成员。然后，着重分析该程序的两个方面。

14.4.1　声明结构数组

声明结构数组和声明其他类型的数组类似。下面是一个声明结构数组的例子：

```
struct book library[MAXBKS];
```

以上代码把 library 声明为一个内含 MAXBKS 个元素的数组。数组的每个元素都是一个 book 类型的数组。因此，library[0]是第 1 个 book 类型的结构变量，library[1]是第 2 个 book 类型的结构变量，以此类推。参看图 14.2 可以帮助读者理解。数组名 library 本身不是结构名，它是一个数组名，该数组中的每个元素都是 struct book 类型的结构变量。

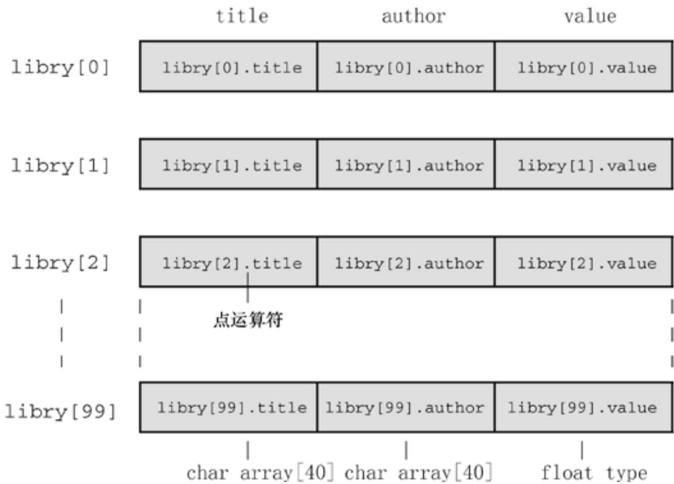


图 14.2　一个结构数组 library[MAXBKS]

14.4.2　标识结构数组的成员

为了标识结构数组中的成员，可以采用访问单独结构的规则：在结构名后面加一个点运算符，再在点运算符后面写上成员名。如下所示：

```
library[0].value  /* 第 1 个数组元素与 value 相关联 */
library[4].title  /* 第 5 个数组元素与 title 相关联 */
```

注意，数组下标紧跟在 library 后面，不是成员名后面：

```
library.value[2]  // 错误
library[2].value  // 正确
```

使用 library[2].value 的原因是：library[2]是结构变量名，正如 library[1]是另一个变量名。

顺带一提，下面的表达式代表什么？

```
library[2].title[4]
```

这是 library 数组第 3 个结构变量（library[2]部分）中书名的第 5 个字符（title[4]部分）。以程序清单 14.2 的输出为例，这个字符是 e。该例指出，点运算符右侧的下标作用于各个成员，点运算符左侧的下标作用与结构数组。

最后，总结一下：

```
library                // 一个 book 结构的数组
library[2]             // 一个数组元素，该元素是 book 结构
library[2].title       // 一个 char 数组（library[2]的 title 成员）
library[2].title[4]    // 数组中 library[2]元素的 title 成员的一个字符
```

下面，我们来讨论一下这个程序。

14.4.3　程序讨论

较之程序清单 14.1，该程序主要的改动之处是：插入一个 while 循环读取多个项。该循环的条件测试是：

```
while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
                         && library[count].title[0] != '\0')
```

表达式 s_gets(library[count].title, MAXTITL)读取一个字符串作为书名，如果 s_gets()尝试读到文件结尾后面，该表达式则返回 NULL。表达式 library[count].title[0] != '\0'判断字符串中的首字符是否是空字符（即，该字符串是否是空字符串）。如果在一行开始处用户按下 Enter 键，相当于输入了一个空字符串，循环将结束。程序中还检查了图书的数量，以免超出数组的大小。

然后，该程序中有如下几行：

```
while (getchar() != '\n')
    continue; /* 清理输入行 */
```

前面章节介绍过，这段代码弥补了 scanf()函数遇到空格和换行符就结束读取的问题。当用户输入书的价格时，可能输入如下信息：

```
12.50[Enter]
```

其传送的字符序列如下：

```
12.50\n
```

scanf()函数接受 1、2、.、5 和 0，但是把\n 留在输入序列中。如果没有上面两行清理输入行的代码，就会把留在输入序列中的换行符当作空行读入，程序以为用户发送了停止输入的信号。我们插入的这

两行代码只会在输入序列中查找并删除\n，不会处理其他字符。这样 s_gets()就可以重新开始下一次输入。

14.5　嵌套结构

有时，在一个结构中包含另一个结构（即嵌套结构）很方便。例如，Shalala Pirosky 创建了一个有关她朋友信息的结构。显然，结构中需要一个成员表示朋友的姓名。然而，名字可以用一个数组来表示，其中包含名和姓这两个成员。程序清单 14.3 是一个简单的示例。

程序清单 14.3　friend.c 程序

```
// friend.c -- 嵌套结构示例
#include <stdio.h>
#define LEN 20
const char * msgs[5] =
{
    "    Thank you for the wonderful evening, ",
    "You certainly prove that a ",
    "is a special kind of guy. We must get together",
    "over a delicious ",
    " and have a few laughs"
};

struct names {                   // 第 1 个结构
    char first[LEN];
    char last[LEN];
};

struct guy {                     // 第 2 个结构
    struct names handle;         // 嵌套结构
    char favfood[LEN];
    char job[LEN];
    float income;
};

int main(void)
{
    struct guy fellow = {        // 初始化一个结构变量
            { "Ewen", "Villard" },
            "grilled salmon",
            "personality coach",
            68112.00
    };

    printf("Dear %s, \n\n", fellow.handle.first);
    printf("%s%s.\n", msgs[0], fellow.handle.first);
    printf("%s%s\n", msgs[1], fellow.job);
    printf("%s\n", msgs[2]);
    printf("%s%s%s", msgs[3], fellow.favfood, msgs[4]);
    if (fellow.income > 150000.0)
        puts("!!");
    else if (fellow.income > 75000.0)
        puts("!");
    else
```

```
        puts(".");
    printf("\n%40s%s\n", " ", "See you soon,");
    printf("%40s%s\n", " ", "Shalala");

    return 0;
}
```

下面是该程序的输出：

```
Dear Ewen,

    Thank you for the wonderful evening, Ewen.
You certainly prove that a personality coach
is a special kind of guy. We must get together
over a delicious grilled salmon and have a few laughs.

                                        See you soon,
                                        Shalala
```

首先，注意如何在结构声明中创建嵌套结构。和声明 int 类型变量一样，进行简单的声明：

```
struct names handle;
```

该声明表明 handle 是一个 struct name 类型的变量。当然，文件中也应包含结构 names 的声明。

其次，注意如何访问嵌套结构的成员，这需要使用两次点运算符：

```
printf("Hello, %s!\n", fellow.handle.first);
```

从左往右解释 fellow.handle.first：

```
(fellow.handle).first
```

也就是说，找到 fellow，然后找到 fellow 的 handle 的成员，再找到 handle 的 first 成员。

14.6　指向结构的指针

喜欢使用指针的人一定很高兴能使用指向结构的指针。至少有 4 个理由可以解释为何要使用指向结构的指针。第一，就像指向数组的指针比数组本身更容易操控（如，排序问题）一样，指向结构的指针通常比结构本身更容易操控。第二，在一些早期的 C 实现中，结构不能作为参数传递给函数，但是可以传递指向结构的指针。第三，即使能传递一个结构，传递指针通常更有效率。第四，一些用于表示数据的结构中包含指向其他结构的指针。

下面的程序（程序清单 14.4）演示了如何定义指向结构的指针和如何用这样的指针访问结构的成员。

程序清单 14.4　friends.c 程序

```
/* friends.c -- 使用指向结构的指针 */
#include <stdio.h>
#define LEN 20

struct names {
    char first[LEN];
    char last[LEN];
};

struct guy {
    struct names handle;
    char favfood[LEN];
    char job[LEN];
```

```
    float income;
};

int main(void)
{
    struct guy fellow[2] = {
            { { "Ewen", "Villard" },
                "grilled salmon",
                "personality coach",
                68112.00
            },
            { { "Rodney", "Swillbelly" },
                "tripe",
                "tabloid editor",
                432400.00
            }
    };
    struct guy * him;      /* 这是一个指向结构的指针 */

    printf("address #1: %p #2: %p\n", &fellow[0], &fellow[1]);
    him = &fellow[0];      /* 告诉编译器该指针指向何处  */
    printf("pointer #1: %p #2: %p\n", him, him + 1);
    printf("him->income is $%.2f: (*him).income is $%.2f\n",
            him->income, (*him).income);
    him++;                 /* 指向下一个结构     */
    printf("him->favfood is %s:  him->handle.last is %s\n",
            him->favfood, him->handle.last);

    return 0;
}
```

该程序的输出如下：

```
address #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
pointer #1: 0x7fff5fbff820 #2: 0x7fff5fbff874
him->income is $68112.00: (*him).income is $68112.00
him->favfood is tripe: him->handle.last is Swillbelly
```

我们先来看如何创建指向 guy 类型结构的指针，然后再分析如何通过该指针指定结构的成员。

14.6.1　声明和初始化结构指针

声明结构指针很简单：

```
struct guy * him;
```

首先是关键字 struct，其次是结构标记 guy，然后是一个星号（*），其后跟着指针名。这个语法和其他指针声明一样。

该声明并未创建一个新的结构，但是指针 him 现在可以指向任意现有的 guy 类型的结构。例如，如果 barney 是一个 guy 类型的结构，可以这样写：

```
him = &barney;
```

和数组不同的是，结构名并不是结构的地址，因此要在结构名前面加上&运算符。

在本例中，fellow 是一个结构数组，这意味着 fellow[0]是一个结构。所以，要让 him 指向 fellow[0]，可以这样写：

```
him = &fellow[0];
```

输出的前两行说明赋值成功。比较这两行发现，him 指向 fellow[0]，him + 1 指向 fellow[1]。注意，him 加 1 相当于 him 指向的地址加 84。在十六进制中，874 - 820 = 54（十六进制）= 84（十进制），因为每个 guy 结构都占用 84 字节的内存：names.first 占用 20 字节，names.last 占用 20 字节，favfood 占用 20 字节，job 占用 20 字节，income 占用 4 字节（假设系统中 float 占用 4 字节）。顺带一提，在有些系统中，一个结构的大小可能大于它各成员大小之和。这是因为系统对数据进行校准的过程中产生了一些“缝隙”。例如，有些系统必须把每个成员都放在偶数地址上，或 4 的倍数的地址上。在这种系统中，结构的内部就存在未使用的“缝隙”。

14.6.2　用指针访问成员

指针 him 指向结构变量 fellow[0]，如何通过 him 获得 fellow[0]的成员的值？程序清单 14.4 中的第 3 行输出演示了两种方法。

第 1 种方法也是最常用的方法：使用->运算符。该运算符由一个连接号（-）后跟一个大于号（>）组成。我们有下面的关系：

```
如果 him == &barney，那么 him->income 即是 barney.income
如果 him == &fellow[0]，那么 him->income 即是 fellow[0].income
```

换句话说，->运算符后面的结构指针和.运算符后面的结构名工作方式相同（不能写成 him.incone，因为 him 不是结构名）。

这里要着重理解 him 是一个指针，但是 hime->income 是该指针所指向结构的一个成员。所以在该例中，him->income 是一个 float 类型的变量。

第 2 种方法是，以这样的顺序指定结构成员的值：如果 him == &fellow[0]，那么 *him == fellow[0]，因为&和*是一对互逆运算符。因此，可以做以下替代：

```
fellow[0].income == (*him).income
```

必须要使用圆括号，因为.运算符比*运算符的优先级高。

总之，如果 him 是指向 guy 类型结构 barney 的指针，下面的关系恒成立：

```
barney.income == (*him).income == him->income // 假设 him == &barney
```

接下来，我们来学习结构和函数的交互。

14.7　向函数传递结构的信息

函数的参数把值传递给函数。每个值都是一个数字——可能是 int 类型、float 类型，可能是 ASCII 字符码，或者是一个地址。然而，一个结构比一个单独的值复杂，所以难怪以前的 C 实现不允许把结构作为参数传递给函数。当前的实现已经移除了这个限制，ANSI C 允许把结构作为参数使用。所以程序员可以选择是传递结构本身，还是传递指向结构的指针。如果你只关心结构中的某一部分，也可以把结构的成员作为参数。我们接下来将分析这 3 种传递方式，首先介绍以结构成员作为参数的情况。

14.7.1　传递结构成员

只要结构成员是一个具有单个值的数据类型（即，int 及其相关类型、char、float、double 或指针），便可把它作为参数传递给接受该特定类型的函数。程序清单 14.5 中的财务分析程序（初级版本）演示了这一点，该程序把客户的银行账户添加到他/她的储蓄和贷款账户中。

程序清单 14.5　funds1.c 程序

```
/* funds1.c -- 把结构成员作为参数传递 */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char     bank[FUNDLEN];
    double   bankfund;
    char     save[FUNDLEN];
    double   savefund;
};

double sum(double, double);

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n",
        sum(stan.bankfund, stan.savefund));
    return 0;
}

/* 两个 double 类型的数相加 */
double sum(double x, double y)
{
    return(x + y);
}
```

运行该程序后输出如下：

```
Stan has a total of $12576.21.
```

看来，这样传递参数没问题。注意，sum()函数既不知道也不关心实际的参数是否是结构的成员，它只要求传入的数据是 double 类型。

当然，如果需要在被调函数中修改主调函数中成员的值，就要传递成员的地址：

```
modify(&stan.bankfund);
```

这是一个更改银行账户的函数。

把结构的信息告诉函数的第 2 种方法是，让被调函数知道自己正在处理一个结构。

14.7.2　传递结构的地址

我们继续解决前面的问题，但是这次把结构的地址作为参数。由于函数要处理 funds 结构，所以必须声明 funds 结构。如程序清单 14.6 所示。

程序清单 14.6　funds2.c 程序

```
/* funds2.c -- 传递指向结构的指针 */
```

```
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char     bank[FUNDLEN];
    double   bankfund;
    char     save[FUNDLEN];
    double   savefund;
};

double sum(const struct funds *);  /* 参数是一个指针 */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n", sum(&stan));

    return 0;
}

double sum(const struct funds * money)
{
    return(money->bankfund + money->savefund);
}
```

运行该程序后输出如下：

```
Stan has a total of $12576.21.
```

sum()函数使用指向 funds 结构的指针（money）作为它的参数。把地址&stan 传递给该函数，使得指针 money 指向结构 stan。然后通过->运算符获取 stan.bankfund 和 stan.savefund 的值。由于该函数不能改变指针所指向值的内容，所以把 money 声明为一个指向 const 的指针。

虽然该函数并未使用其他成员，但是也可以访问它们。注意，必须使用&运算符来获取结构的地址。和数组名不同，结构名只是其地址的别名。

14.7.3 传递结构

对于允许把结构作为参数的编译器，可以把程序清单 14.6 重写为程序清单 14.7。

程序清单 14.7 funds3.c 程序

```
/* funds3.c -- 传递一个结构 */
#include <stdio.h>
#define FUNDLEN 50

struct funds {
    char   bank[FUNDLEN];
    double bankfund;
    char   save[FUNDLEN];
```

```
    double savefund;
};

double sum(struct funds moolah);  /* 参数是一个结构 */

int main(void)
{
    struct funds stan = {
        "Garlic-Melon Bank",
        4032.27,
        "Lucky's Savings and Loan",
        8543.94
    };

    printf("Stan has a total of $%.2f.\n", sum(stan));

    return 0;
}

double sum(struct funds moolah)
{
    return(moolah.bankfund + moolah.savefund);
}
```

下面是运行该程序后的输出：

```
Stan has a total of $12576.21.
```

该程序把程序清单 14.6 中指向 struct funds 类型的结构指针 money 替换成 struct funds 类型的结构变量 moolah。调用 sum()时，编译器根据 funds 模板创建了一个名为 moolah 的自动结构变量。然后，该结构的各成员被初始化为 stan 结构变量相应成员的值的副本。因此，程序使用原来结构的副本进行计算，然而，传递指针的程序清单 14.6 使用的是原始的结构进行计算。由于 moolah 是一个结构，所以该程序使用 moolah.bankfund，而不是 moolah->bankfund。另一方面，由于 money 是指针，不是结构，所以程序清单 14.6 使用的是 monet->bankfund。

14.7.4 其他结构特性

现在的 C 允许把一个结构赋值给另一个结构，但是数组不能这样做。也就是说，如果 n_data 和 o_data 都是相同类型的结构，可以这样做：

```
o_data = n_data;  // 把一个结构赋值给另一个结构
```

这条语句把 n_data 的每个成员的值都赋给 o_data 的相应成员。即使成员是数组，也能完成赋值。另外，还可以把一个结构初始化为相同类型的另一个结构：

```
struct names right_field = {"Ruthie", "George"};
struct names captain = right_field; // 把一个结构初始化为另一个结构
```

现在的 C（包括 ANSI C），函数不仅能把结构本身作为参数传递，还能把结构作为返回值返回。把结构作为函数参数可以把结构的信息传送给函数；把结构作为返回值的函数能把结构的信息从被调函数传回主调函数。结构指针也允许这种双向通信，因此可以选择任一种方法来解决编程问题。我们通过另一组程序示例来演示这两种方法。

为了对比这两种方法，我们先编写一个程序以传递指针的方式处理结构，然后以传递结构和返回结构的方式重写该程序。

程序清单 14.8 names1.c 程序

```
/* names1.c -- 使用指向结构的指针 */
#include <stdio.h>
#include <string.h>

#define NLEN 30
struct namect {
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

void getinfo(struct namect *);
void makeinfo(struct namect *);
void showinfo(const struct namect *);
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    return 0;
}

void getinfo(struct namect * pst)
{
    printf("Please enter your first name.\n");
    s_gets(pst->fname, NLEN);
    printf("Please enter your last name.\n");
    s_gets(pst->lname, NLEN);
}

void makeinfo(struct namect * pst)
{
    pst->letters = strlen(pst->fname) +strlen(pst->lname);
}

void showinfo(const struct namect * pst)
{
    printf("%s %s, your name contains %d letters.\n",
            pst->fname, pst->lname, pst->letters);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
```

```
        find = strchr(st, '\n');    // 查找换行符
        if (find)                   // 如果地址不是 NULL,
            *find = '\0';           // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;           // 处理输入行的剩余字符
    }
    return ret_val;
}
```

下面是编译并运行该程序后的一个输出示例：

```
Please enter your first name.
Viola
Please enter your last name.
Plunderfest
Viola Plunderfest, your name contains 16 letters.
```

该程序把任务分配给 3 个函数来完成，都在 main()中调用。每调用一个函数就把 person 结构的地址传递给它。

getinfo()函数把结构的信息从自身传递给 main()。该函数通过与用户交互获得姓名，并通过 pst 指针定位，将其放入 person 结构中。由于 pst->lname 意味着 pst 指向结构的 lname 成员，这使得 pst->lname 等价于 char 数组的名称，因此做 s_gets()的参数很合适。注意，虽然 getinfo()给 main()提供了信息，但是它并未使用返回机制，所以其返回类型是 void。

makeinfo()函数使用双向传输方式传送信息。通过使用指向 person 的指针，该指针定位了储存在该结构中的名和姓。该函数使用 C 库函数 strlen()分别计算名和姓中的字母总数，然后使用 person 的地址储存两数之和。同样，makeinfo()函数的返回类型也是 void。

showinfo()函数使用一个指针定位待打印的信息。因为该函数不改变数组的内容，所以将其声明为 const。

所有这些操作中，只有一个结构变量 person，每个函数都使用该结构变量的地址来访问它。一个函数把信息从自身传回主调函数，一个函数把信息从主调函数传给自身，一个函数通过双向传输来传递信息。

现在，我们来看如何使用结构参数和返回值来完成相同的任务。第一，为了传递结构本身，函数的参数必须是 person，而不是&person。那么，相应的形式参数应声明为 struct namect，而不是指向该类型的指针。第二，可以通过返回一个结构，把结构的信息返回给 main()。程序清单 14.9 演示了不使用指针的版本。

程序清单 14.9　names2.c 程序

```
/* names2.c -- 传递并返回结构 */
#include <stdio.h>
#include <string.h>

#define NLEN 30
struct namect {
    char fname[NLEN];
    char lname[NLEN];
    int letters;
};

struct namect getinfo(void);
struct namect makeinfo(struct namect);
```

```
void showinfo(struct namect);
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    person = getinfo();
    person = makeinfo(person);
    showinfo(person);

    return 0;
}

struct namect getinfo(void)
{
    struct namect temp;
    printf("Please enter your first name.\n");
    s_gets(temp.fname, NLEN);
    printf("Please enter your last name.\n");
    s_gets(temp.lname, NLEN);

    return temp;
}

struct namect makeinfo(struct namect info)
{
    info.letters = strlen(info.fname) + strlen(info.lname);

    return info;
}

void showinfo(struct namect info)
{
    printf("%s %s, your name contains %d letters.\n",
            info.fname, info.lname, info.letters);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;          // 处理输入行的剩余部分
    }
    return ret_val;
}
```

该版本最终的输出和前面版本相同，但是它使用了不同的方式。程序中的每个函数都创建了自己的 person 备份，所以该程序使用了 4 个不同的结构，不像前面的版本只使用一个结构。

例如，考虑 makeinfo()函数。在第 1 个程序中，传递的是 person 的地址，该函数实际上处理的是 person 的值。在第 2 个版本的程序中，创建了一个新的结构 info。储存在 person 中的值被拷贝到 info 中，函数处理的是这个副本。因此，统计完字母个数后，计算结果储存在 info 中，而不是 person 中。然而，返回机制弥补了这一点。makeinfo()中的这行代码：

```
return info;
```

与 main()中的这行结合：

```
person = makeinfo(person);
```

把储存在 info 中的值拷贝到 person 中。注意，必须把 makeinfo()函数声明为 struct namect 类型，所以该函数要返回一个结构。

14.7.5　结构和结构指针的选择

假设要编写一个与结构相关的函数，是用结构指针作为参数，还是用结构作为参数和返回值？两者各有优缺点。

把指针作为参数有两个优点：无论是以前还是现在的 C 实现都能使用这种方法，而且执行起来很快，只需要传递一个地址。缺点是无法保护数据。被调函数中的某些操作可能会意外影响原来结构中的数据。不过，ANSI C 新增的 const 限定符解决了这个问题。例如，如果在程序清单 14.8 中，showinfo()函数中的代码改变了结构的任意成员，编译器会捕获这个错误。

把结构作为参数传递的优点是，函数处理的是原始数据的副本，这保护了原始数据。另外，代码风格也更清楚。假设定义了下面的结构类型：

```
struct vector {double x; double y;};
```

如果用 vector 类型的结构 ans 储存相同类型结构 a 和 b 的和，就要把结构作为参数和返回值：

```
struct vector ans, a, b;
struct vector sum_vect(struct vector, struct vector);
...
ans = sum_vect(a,b);
```

对程序员而言，上面的版本比用指针传递的版本更自然。指针版本如下：

```
struct vector ans, a, b;
void sum_vect(const struct vector *, const struct vector *, struct vector *);
...
sum_vect(&a, &b, &ans);
```

另外，如果使用指针版本，程序员必须记住总和的地址应该是第 1 个参数还是第 2 个参数的地址。

传递结构的两个缺点是：较老版本的实现可能无法处理这样的代码，而且传递结构浪费时间和存储空间。尤其是把大型结构传递给函数，而它只使用结构中的一两个成员时特别浪费。这种情况下传递指针或只传递函数所需的成员更合理。

通常，程序员为了追求效率会使用结构指针作为函数参数，如需防止原始数据被意外修改，使用 const 限定符。按值传递结构是处理小型结构最常用的方法。

14.7.6　结构中的字符数组和字符指针

到目前为止，我们在结构中都使用字符数组来储存字符串。是否可以使用指向 char 的指针来代替字符数组？例如，程序清单 14.3 中有如下声明：

```
#define LEN 20
struct names {
    char first[LEN];
    char last[LEN];
};
```

其中的结构声明是否可以这样写：

```
struct pnames {
    char * first;
    char * last;
};
```

当然可以，但是如果不理解这样做的含义，可能会有麻烦。考虑下面的代码：

```
struct names veep = {"Talia", "Summers"};
struct pnames treas = {"Brad", "Fallingjaw"};
printf("%s and %s\n", veep.first, treas.first);
```

以上代码都没问题，也能正常运行，但是思考一下字符串被储存在何处。对于 `struct names` 类型的结构变量 `veep`，以上字符串都储存在结构内部，结构总共要分配 40 字节储存姓名。然而，对于 `struct pnames` 类型的结构变量 `treas`，以上字符串储存在编译器储存常量的地方。结构本身只储存了两个地址，在我们的系统中共占 16 字节。尤其是，`struct pnames` 结构不用为字符串分配任何存储空间。它使用的是储存在别处的字符串（如，字符串常量或数组中的字符串）。简而言之，在 `pnames` 结构变量中的指针应该只用来在程序中管理那些已分配和在别处分配的字符串。

我们看看这种限制在什么情况下出问题。考虑下面的代码：

```
struct names accountant;
struct pnames attorney;
puts("Enter the last name of your accountant:");
scanf("%s", accountant.last);
puts("Enter the last name of your attorney:");
scanf("%s", attorney.last);     /* 这里有一个潜在的危险 */
```

就语法而言，这段代码没问题。但是，用户的输入储存到哪里去了？对于会计师（*accountant*），他的名储存在 `accountant` 结构变量的 `last` 成员中，该结构中有一个储存字符串的数组。对于律师（*attorney*），`scanf()` 把字符串放到 `attorney.last` 表示的地址上。由于这是未经初始化的变量，地址可以是任何值，因此程序可以把名放在任何地方。如果走运的话，程序不会出问题，至少暂时不会出问题，否则这一操作会导致程序崩溃。实际上，如果程序能正常运行并不是好事，因为这意味着一个未被觉察的危险潜伏在程序中。

因此，如果要用结构储存字符串，用字符数组作为成员比较简单。用指向 `char` 的指针也行，但是误用会导致严重的问题。

14.7.7 结构、指针和 malloc()

如果使用 `malloc()` 分配内存并使用指针储存该地址，那么在结构中使用指针处理字符串就比较合理。这种方法的优点是，可以请求 `malloc()` 为字符串分配合适的存储空间。可以要求用 4 字节储存`"Joe"`和用 18 字节储存`"Rasolofomasoandro"`。用这种方法改写程序清单 14.9 并不费劲。主要是更改结构声明（用指针代替数组）和提供一个新版本的 `getinfo()` 函数。新的结构声明如下：

```
struct namect {
    char * fname; // 用指针代替数组
    char * lname;
    int letters;
};
```

新版本的 getinfo()把用户的输入读入临时数组中，调用 malloc()函数分配存储空间，并把字符串拷贝到新分配的存储空间中。对名和姓都要这样做：

```
void getinfo (struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);
    // 分配内存储存名
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // 把名拷贝到已分配的内存
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}
```

要理解这两个字符串都未储存在结构中，它们储存在 malloc()分配的内存块中。然而，结构中储存着这两个字符串的地址，处理字符串的函数通常都要使用字符串的地址。因此，不用修改程序中的其他函数。

第 12 章建议，应该成对使用 malloc()和 free()。因此，还要在程序中添加一个新的函数 cleanup()，用于释放程序动态分配的内存。如程序清单 14.10 所示。

程序清单 14.10　names3.c 程序

```
// names3.c -- 使用指针和 malloc()
#include <stdio.h>
#include <string.h>   // 提供 strcpy()、strlen() 的原型
#include <stdlib.h>   // 提供 malloc()、free() 的原型
#define SLEN 81
struct namect {
    char * fname;  // 使用指针
    char * lname;
    int letters;
};

void getinfo(struct namect *);      // 分配内存
void makeinfo(struct namect *);
void showinfo(const struct namect *);
void cleanup(struct namect *);      // 调用该函数时释放内存
char * s_gets(char * st, int n);

int main(void)
{
    struct namect person;

    getinfo(&person);
    makeinfo(&person);
    showinfo(&person);
    cleanup(&person);

    return 0;
}
```

```
void getinfo(struct namect * pst)
{
    char temp[SLEN];
    printf("Please enter your first name.\n");
    s_gets(temp, SLEN);
    // 分配内存以储存名
    pst->fname = (char *) malloc(strlen(temp) + 1);
    // 把名拷贝到动态分配的内存中
    strcpy(pst->fname, temp);
    printf("Please enter your last name.\n");
    s_gets(temp, SLEN);
    pst->lname = (char *) malloc(strlen(temp) + 1);
    strcpy(pst->lname, temp);
}

void makeinfo(struct namect * pst)
{
    pst->letters = strlen(pst->fname) +
        strlen(pst->lname);
}

void showinfo(const struct namect * pst)
{
    printf("%s %s, your name contains %d letters.\n",
        pst->fname, pst->lname, pst->letters);
}

void cleanup(struct namect * pst)
{
    free(pst->fname);
    free(pst->lname);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;          // 处理输入行的剩余部分
    }
    return ret_val;
}
```

下面是该程序的输出：

```
Please enter your first name.
Floresiensis
Please enter your last name.
Mann
Floresiensis Mann, your name contains 16 letters.
```

14.7.8　复合字面量和结构（C99）

C99 的复合字面量特性可用于结构和数组。如果只需要一个临时结构值，复合字面量很好用。例如，可以使用复合字面量创建一个数组作为函数的参数或赋给另一个结构。语法是把类型名放在圆括号中，后面紧跟一个用花括号括起来的初始化列表。例如，下面是 `struct book` 类型的复合字面量：

```
(struct book) {"The Idiot", "Fyodor Dostoyevsky", 6.99}
```

程序清单 14.11 中的程序示例，使用复合字面量为一个结构变量提供两个可替换的值（在撰写本书时，并不是所有的编译器都支持这个特性，不过这是时间的问题）。

程序清单 14.11　complit.c 程序

```
/* complit.c -- 复合字面量 */
#include <stdio.h>
#define MAXTITL  41
#define MAXAUTL  31

struct book {         // 结构模版：标记是 book
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book readfirst;
    int score;

    printf("Enter test score: ");
    scanf("%d", &score);

    if (score >= 84)
        readfirst = (struct book) {"Crime and Punishment",
                                 "Fyodor Dostoyevsky",
                                 11.25};
    else
        readfirst = (struct book) {"Mr. Bouncy's Nice Hat",
                                 "Fred Winsome",
                                 5.99};

    printf("Your assigned reading:\n");
    printf("%s by %s: $%.2f\n", readfirst.title,
        readfirst.author, readfirst.value);

    return 0;
}
```

还可以把复合字面量作为函数的参数。如果函数接受一个结构，可以把复合字面量作为实际参数传递：

```
struct rect {double x; double y;};
double rect_area(struct rect r){return r.x * r.y;}
...
double area;
area = rect_area( (struct rect) {10.5, 20.0});
```

值 210 被赋给 area。

如果函数接受一个地址，可以传递复合字面量的地址：

```
struct rect {double x; double y;};
double rect_areap(struct rect * rp){return rp->x * rp->y;}
...
double area;
area = rect_areap( &(struct rect) {10.5, 20.0});
```

值 210 被赋给 area。

复合字面量在所有函数的外部，具有静态存储期；如果复合字面量在块中，则具有自动存储期。复合字面量和普通初始化列表的语法规则相同。这意味着，可以在复合字面量中使用指定初始化器。

14.7.9　伸缩型数组成员（C99）

C99 新增了一个特性：**伸缩型数组成员**（*flexible array member*），利用这项特性声明的结构，其最后一个数组成员具有一些特性。第 1 个特性是，该数组不会立即存在。第 2 个特性是，使用这个伸缩型数组成员可以编写合适的代码，就好像它确实存在并具有所需数目的元素一样。这可能听起来很奇怪，所以我们来一步步地创建和使用一个带伸缩型数组成员的结构。

首先，声明一个伸缩型数组成员有如下规则：

- 伸缩型数组成员必须是结构的最后一个成员；
- 结构中必须至少有一个成员；
- 伸缩数组的声明类似于普通数组，只是它的方括号中是空的。

下面用一个示例来解释以上几点：

```
struct flex
{
    int count;
    double average;
    double scores[];   // 伸缩型数组成员
};
```

声明一个 struct flex 类型的结构变量时，不能用 scores 做任何事，因为没有给这个数组预留存储空间。实际上，C99 的意图并不是让你声明 struct flex 类型的变量，而是希望你声明一个指向 struct flex 类型的**指针**，然后用 malloc()来分配足够的空间，以储存 struct flex 类型结构的常规内容和伸缩型数组成员所需的额外空间。例如，假设用 scores 表示一个内含 5 个 double 类型值的数组，可以这样做：

```
struct flex * pf; // 声明一个指针
// 请求为一个结构和一个数组分配存储空间
pf = malloc(sizeof(struct flex) + 5 * sizeof(double));
```

现在有足够的存储空间储存 count、average 和一个内含 5 个 double 类型值的数组。可以用指针 pf 访问这些成员：

```
pf->count = 5;         // 设置 count 成员
pf->scores[2] = 18.5;  // 访问数组成员的一个元素
```

程序清单 14.13 进一步扩展了这个例子，让伸缩型数组成员在第 1 种情况下表示 5 个值，在第 2 种情况下代表 9 个值。该程序也演示了如何编写一个函数处理带伸缩型数组元素的结构。

程序清单 14.12　flexmemb.c 程序

```
// flexmemb.c -- 伸缩型数组成员（C99 新增特性）
#include <stdio.h>
#include <stdlib.h>

struct flex
{
    size_t count;
    double average;
    double scores []; // 伸缩型数组成员
};

void showFlex(const struct flex * p);

int main(void)
{
    struct flex * pf1, *pf2;
    int n = 5;
    int i;
    int tot = 0;

    // 为结构和数组分配存储空间
    pf1 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf1->count = n;
    for (i = 0; i < n; i++)
    {
        pf1->scores[i] = 20.0 - i;
        tot += pf1->scores[i];
    }
    pf1->average = tot / n;
    showFlex(pf1);

    n = 9;
    tot = 0;
    pf2 = malloc(sizeof(struct flex) + n * sizeof(double));
    pf2->count = n;
    for (i = 0; i < n; i++)
    {
        pf2->scores[i] = 20.0 - i / 2.0;
        tot += pf2->scores[i];
    }
    pf2->average = tot / n;
    showFlex(pf2);
    free(pf1);
    free(pf2);

    return 0;
}

void showFlex(const struct flex * p)
{
```

```
    int i;
    printf("Scores : ");
    for (i = 0; i < p->count; i++)
        printf("%g ", p->scores[i]);
    printf("\nAverage: %g\n", p->average);
}
```

下面是该程序的输出：

```
Scores : 20 19 18 17 16
Average: 18
Scores : 20 19.5 19 18.5 18 17.5 17 16.5 16
Average: 17
```

带伸缩型数组成员的结构确实有一些特殊的处理要求。第一，不能用结构进行赋值或拷贝：

```
struct flex * pf1, *pf2;    // *pf1 和*pf2 都是结构
...
*pf2 = *pf1;                // 不要这样做
```

这样做只能拷贝除伸缩型数组成员以外的其他成员。确实要进行拷贝，应使用 `memcpy()`函数（第 16 章中介绍）。

第二，不要以按值方式把这种结构传递给结构。原因相同，按值传递一个参数与赋值类似。要把结构的地址传递给函数。

第三，不要使用带伸缩型数组成员的结构作为数组成员或另一个结构的成员。

这种类似于在结构中最后一个成员是伸缩型数组的情况，称为 struct hack。除了伸缩型数组成员在声明时用空的方括号外，struct hack 特指大小为 `0` 的数组。然而，struct hack 是针对特殊编译器（GCC）的，不属于 C 标准。这种伸缩型数组成员方法是标准认可的编程技巧。

14.7.10　匿名结构（C11）

匿名结构是一个没有名称的结构成员。为了理解它的工作原理，我们先考虑如何创建嵌套结构：

```
struct names
{
    char first[20];
    char last[20];
};
struct person
{
    int id;
    struct names name; // 嵌套结构成员
};
struct person ted = {8483, {"Ted", "Grass"}};
```

这里，`name` 成员是一个嵌套结构，可以通过类似 `ted.name.first` 的表达式访问`"ted"`：

```
puts(ted.name.first);
```

在 C11 中，可以用嵌套的匿名成员结构定义 `person`：

```
struct person
{
    int id;
    struct {char first[20]; char last[20];}; // 匿名结构
};
```

初始化 `ted` 的方式相同：

```
struct person ted = {8483, {"Ted", "Grass"}};
```

但是，在访问 ted 时简化了步骤，只需把 first 看作是 person 的成员那样使用它：

```
puts(ted.first);
```

当然，也可以把 first 和 last 直接作为 person 的成员，删除嵌套循环。匿名特性在嵌套联合中更加有用，我们在本章后面介绍。

14.7.11　使用结构数组的函数

假设一个函数要处理一个结构数组。由于数组名就是该数组的地址，所以可以把它传递给函数。另外，该函数还需访问结构模板。为了理解该函数的工作原理，程序清单 14.13 把前面的金融程序扩展为两人，所以需要一个内含两个 funds 结构的数组。

程序清单 14.13　funds4.c 程序

```
/* funds4.c -- 把结构数组传递给函数 */
#include <stdio.h>
#define FUNDLEN 50
#define N 2

struct funds {
    char     bank[FUNDLEN];
    double   bankfund;
    char save[FUNDLEN];
    double   savefund;
};

double sum(const struct funds money [], int n);

int main(void)
{
    struct funds jones[N] = {
            {
                "Garlic-Melon Bank",
                4032.27,
                "Lucky's Savings and Loan",
                8543.94

            },
            {
                "Honest Jack's Bank",
                3620.88,
                "Party Time Savings",
                3802.91
            }
    };

    printf("The Joneses have a total of $%.2f.\n",sum(jones, N));

    return 0;
}

double sum(const struct funds money [], int n)
{
    double total;
```

```
    int i;

    for (i = 0, total = 0; i < n; i++)
        total += money[i].bankfund + money[i].savefund;

    return(total);
}
```

该程序的输出如下：

```
The Joneses have a total of $20000.00.
```

（读者也许认为这个总和有些巧合！）

数组名 jones 是该数组的地址，即该数组首元素（jones[0]）的地址。因此，指针 money 的初始值相当于通过下面的表达式获得：

```
money = &jones[0];
```

因为 money 指向 jones 数组的首元素，所以 money[0]是该数组的另一个名称。与此类似，money[1]是第 2 个元素。每个元素都是一个 funds 类型的结构，所以都可以使用点运算符（.）来访问 funds 类型结构的成员。

下面是几个要点。

- 可以把数组名作为数组中第 1 个结构的地址传递给函数。
- 然后可以用数组表示法访问数组中的其他结构。注意下面的函数调用与使用数组名效果相同：

  ```
  sum(&jones[0], N)
  ```

  因为 jones 和&jones[0]的地址相同，使用数组名是传递结构地址的一种间接的方法。
- 由于 sum()函数不能改变原始数据，所以该函数使用了 ANSI C 的限定符 const。

14.8　把结构内容保存到文件中

由于结构可以储存不同类型的信息，所以它是构建数据库的重要工具。例如，可以用一个结构储存雇员或汽车零件的相关信息。最终，我们要把这些信息储存在文件中，并且能再次检索。数据库文件可以包含任意数量的此类数据对象。储存在一个结构中的整套信息被称为**记录**（*record*），单独的项被称为**字段**（*field*）。本节我们来探讨这个主题。

或许储存记录最没效率的方法是用 fprintf()。例如，回忆程序清单 14.1 中的 book 结构：

```
#define MAXTITL 40
#define MAXAUTL 40
struct book {
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};
```

如果 pbook 标识一个文件流，那么通过下面这条语句可以把信息储存在 struct book 类型的结构变量 primer 中：

```
fprintf(pbooks, "%s %s %.2f\n", primer.title,primer.author, primer.value);
```

对于一些结构（如，有 30 个成员的结构），这个方法用起来很不方便。另外，在检索时还存在问题，因为程序要知道一个字段结束和另一个字段开始的位置。虽然用固定字段宽度的格式可以解决这个问题（例如，"%39s%39s%8.2f"），但是这个方法仍然很笨拙。

更好的方案是使用 fread()和 fwrite()函数读写结构大小的单元。回忆一下，这两个函数使用与程序相同的二进制表示法。例如：

```
fwrite(&primer, sizeof(struct book), 1, pbooks);
```

定位到 primer 结构变量开始的位置，并把结构中所有的字节都拷贝到与 pbooks 相关的文件中。sizeof(struct book)告诉函数待拷贝的一块数据的大小，1 表明一次拷贝一块数据。带相同参数的 fread()函数从文件中拷贝一块结构大小的数据到&primer 指向的位置。简而言之，这两个函数一次读写整个记录，而不是一个字段。

以二进制表示法储存数据的缺点是，不同的系统可能使用不同的二进制表示法，所以数据文件可能不具可移植性。甚至同一个系统，不同编译器设置也可能导致不同的二进制布局。

14.8.1　保存结构的程序示例

为了演示如何在程序中使用这些函数，我们把程序清单 14.2 修改为一个新的版本（即程序清单 14.14），把书名保存在 book.dat 文件中。如果该文件已存在，程序将显示它当前的内容，然后允许在文件中添加内容（如果你使用的是早期的 Borland 编译器，请参阅程序清单 14.2 后面的“Borland C 和浮点数”）。

程序清单 14.14　booksave.c 程序

```
/* booksave.c -- 在文件中保存结构中的内容 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXTITL  40
#define MAXAUTL  40
#define MAXBKS   10            /* 最大书籍数量 */
char * s_gets(char * st, int n);
struct book {                  /* 建立 book 模板 */
    char title[MAXTITL];
    char author[MAXAUTL];
    float value;
};

int main(void)
{
    struct book library[MAXBKS]; /* 结构数组  */
    int count = 0;
    int index, filecount;
    FILE * pbooks;
    int size = sizeof(struct book);

    if ((pbooks = fopen("book.dat", "a+b")) == NULL)
    {
        fputs("Can't open book.dat file\n", stderr);
        exit(1);
    }

    rewind(pbooks);            /* 定位到文件开始 */
    while (count < MAXBKS &&  fread(&library[count], size,
        1, pbooks) == 1)
    {
        if (count == 0)
```

```
            puts("Current contents of book.dat:");
        printf("%s by %s: $%.2f\n", library[count].title,
            library[count].author, library[count].value);
        count++;
    }
    filecount = count;
    if (count == MAXBKS)
    {
        fputs("The book.dat file is full.", stderr);
        exit(2);
    }

    puts("Please add new book titles.");
    puts("Press [enter] at the start of a line to stop.");
    while (count < MAXBKS && s_gets(library[count].title, MAXTITL) != NULL
        && library[count].title[0] != '\0')
    {
        puts("Now enter the author.");
        s_gets(library[count].author, MAXAUTL);
        puts("Now enter the value.");
        scanf("%f", &library[count++].value);
        while (getchar() != '\n')
            continue;          /* 清理输入行 */
        if (count < MAXBKS)
            puts("Enter the next title.");
    }

    if (count > 0)
    {
        puts("Here is the list of your books:");
        for (index = 0; index < count; index++)
            printf("%s by %s: $%.2f\n", library[index].title,
            library[index].author, library[index].value);
        fwrite(&library[filecount], size, count - filecount,
            pbooks);
    }
    else
        puts("No books? Too bad.\n");

    puts("Bye.\n");
    fclose(pbooks);

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)              // 如果地址不是 NULL,
```

```
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;      // 清理输入行
    }
    return ret_val;
}
```

我们先看几个运行示例，然后再讨论程序中的要点。

```
$ booksave
Please add new book titles.
Press [enter] at the start of a line to stop.
Metric Merriment
Now enter the author.
Polly Poetica
Now enter the value.
18.99
Enter the next title.
Deadly Farce
Now enter the author.
Dudley Forse
Now enter the value.
15.99
Enter the next title.
[enter]
Here is the list of your books:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Bye.
$ booksave
Current contents of book.dat:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
Please add new book titles.
The Third Jar
Now enter the author.
Nellie Nostrum
Now enter the value.
22.99
Enter the next title.
[enter]
Here is the list of your books:
Metric Merriment by Polly Poetica: $18.99
Deadly Farce by Dudley Forse: $15.99
The Third Jar by Nellie Nostrum: $22.99
Bye.
$
```

再次运行 booksave.c 程序把这 3 本书作为当前的文件记录打印出来。

14.8.2　程序要点

首先，以"a+b"模式打开文件。a+部分允许程序读取整个文件并在文件的末尾添加内容。b 是 ANSI 的一种标识方法，表明程序将使用二进制文件格式。对于不接受 b 模式的 UNIX 系统，可以省略 b，因为

UNIX 只有一种文件形式。对于早期的 ANSI 实现，要找出和 b 等价的表示法。

我们选择二进制模式是因为`fread()`和`fwrite()`函数要使用二进制文件。虽然结构中有些内容是文本，但是`value`成员不是文本。如果使用文本编辑器查看`book.dat`，该结构本文部分的内容显示正常，但是数值部分的内容不可读，甚至会导致文本编辑器出现乱码。

`rewrite()`函数确保文件指针位于文件开始处，为读文件做好准备。

第 1 个`while`循环每次把一个结构读到结构数组中，当数组已满或读完文件时停止。变量`filecount`统计已读结构的数量。

第 2 个`while`按下循环提示用户进行输入，并接受用户的输入。和程序清单 14.2 一样，当数组已满或用户在一行的开始处按下 Enter 键时，循环结束。注意，该循环开始时`count`变量的值是第 1 个循环结束后的值。该循环把新输入项添加到数组的末尾。

然后`for`循环打印文件和用户输入的数据。因为该文件是以附加模式打开，所以新写入的内容添加到文件现有内容的末尾。

我们本可以用一个循环在文件末尾一次添加一个结构，但还是决定用`fwrite()`一次写入一块数据。对表达式`count - filecount`求值得新添加的书籍数量，然后调用`fwrite()`把结构大小的块写入文件。由于表达式`&library[filecount]`是数组中第 1 个新结构的地址，所以拷贝就从这里开始。

也许该例是把结构写入文件和检索它们的最简单的方法，但是这种方法浪费存储空间，因为这还保存了结构中未使用的部分。该结构的大小是`2×40×sizeof(char)+sizeof(float)`，在我们的系统中共 84 字节。实际上不是每个输入项都需要这么多空间。但是，让每个输入块的大小相同在检索数据时很方便。

另一个方法是使用可变大小的记录。为了方便读取文件中的这种记录，每个记录以数值字段规定记录的大小。这比上一种方法复杂。通常，这种方法涉及接下来要介绍的“链式结构”和第 16 章的动态内存分配。

14.9　链式结构

在结束讨论结构之前，我们想简要介绍一下结构的多种用途之一：创建新的数据形式。计算机用户已经开发出的一些数据形式比我们提到过的数组和简单结构更有效地解决特定的问题。这些形式包括队列、二叉树、堆、哈希表和图表。许多这样的形式都由**链式结构**（*linked structure*）组成。通常，每个结构都包含一两个数据项和一两个指向其他同类型结构的指针。这些指针把一个结构和另一个结构链接起来，并提供一种路径能遍历整个彼此链接的结构。例如，图 14.3 演示了一个二叉树结构，每个单独的结构（或节点）都和它下面的两个结构（或节点）相连。

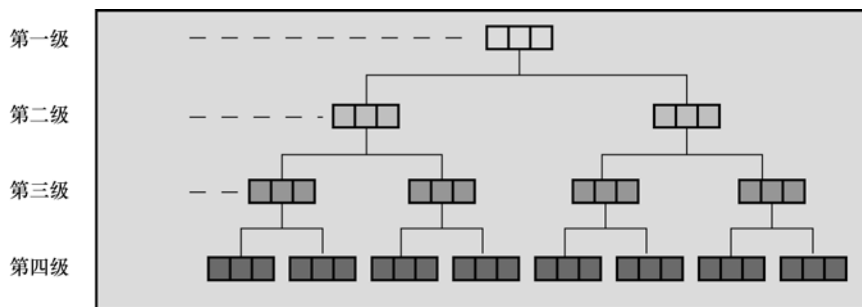


图 14.3　一个二叉树结构

图 14.3 中显示的分级或树状的结构是否比数组高效？考虑一个有 10 级节点的树的情况。它有 $2^{10}-1$（或 1023）个节点，可以储存 1023 个单词。如果这些单词以某种规则排列，那么可以从最顶层开始，逐级向下

移动查找单词，最多只需移动 9 次便可找到任意单词。如果把这些单词都放在一个数组中，最多要查找 1023 个元素才能找出所需的单词。

如果你对这些高级概念感兴趣，可以阅读一些关于数据结构的书籍。使用 C 结构，可以创建和使用那些书中介绍的各种数据形式。另外，第 17 章中也介绍了一些高级数据形式。

本章对结构的概念介绍至此为止，第 17 章中会给出链式结构的例子。下面，我们介绍 C 语言中的联合、枚举和 typedef。

14.10　联合简介

联合（*union*）是一种数据类型，它能在同一个内存空间中储存不同的数据类型（不是同时储存）。其典型的用法是，设计一种表以储存既无规律、事先也不知道顺序的混合类型。使用联合类型的数组，其中的联合都大小相等，每个联合可以储存各种数据类型。

创建联合和创建结构的方式相同，需要一个联合模板和联合变量。可以用一个步骤定义联合，也可以用联合标记分两步定义。下面是一个带标记的联合模板：

```
union hold {
    int digit;
    double bigfl;
    char letter;
};
```

根据以上形式声明的结构可以储存一个 int 类型、一个 double 类型和 char 类型的值。然而，声明的联合只能储存一个 int 类型的值或一个 double 类型的值或 char 类型的值。

下面定义了 3 个与 hold 类型相关的变量：

```
union hold fit;        // hold 类型的联合变量
union hold save[10];   // 内含 10 个联合变量的数组
union hold * pu;       // 指向 hold 类型联合变量的指针
```

第 1 个声明创建了一个单独的联合变量 fit。编译器分配足够的空间以便它能储存联合声明中占用最大字节的类型。在本例中，占用空间最大的是 double 类型的数据。在我们的系统中，double 类型占 64 位，即 8 字节。第 2 个声明创建了一个数组 save，内含 10 个元素，每个元素都是 8 字节。第 3 个声明创建了一个指针，该指针变量储存 hold 类型联合变量的地址。

可以初始化联合。需要注意的是，联合只能储存一个值，这与结构不同。有 3 种初始化的方法：把一个联合初始化为另一个同类型的联合；初始化联合的第 1 个元素；或者根据 C99 标准，使用指定初始化器：

```
union hold valA;
valA.letter = 'R';
union hold valB = valA;              // 用另一个联合来初始化
union hold valC = {88};              // 初始化联合的 digit 成员
union hold valD = {.bigfl = 118.2}; // 指定初始化器
```

14.10.1　使用联合

下面是联合的一些用法：

```
fit.digit = 23;   //把 23 储存在 fit，占 2 字节
fit.bigfl = 2.0;  // 清除 23，储存 2.0，占 8 字节
fit.letter = 'h'; // 清除 2.0，储存 h，占 1 字节
```

点运算符表示正在使用哪种数据类型。在联合中，一次只储存一个值。即使有足够的空间，也不能同

时储存一个 char 类型值和一个 int 类型值。编写代码时要注意当前储存在联合中的数据类型。

和用指针访问结构使用->运算符一样，用指针访问联合时也要使用->运算符：

```
pu = &fit;
x = pu->digit; // 相当于 x = fit.digit
```

不要像下面的语句序列这样：

```
fit.letter = 'A';
flnum = 3.02*fit.bigfl; // 错误
```

以上语句序列是错误的，因为储存在 fit 中的是 char 类型，但是下一行却假定 fit 中的内容是 double 类型。

不过，用一个成员把值储存在一个联合中，然后用另一个成员查看内容，这种做法有时很有用。下一章的程序清单 15.4 就给出了一个这样的例子。

联合的另一种用法是，在结构中储存与其成员有从属关系的信息。例如，假设用一个结构表示一辆汽车。如果汽车属于驾驶者，就要用一个结构成员来描述这个所有者。如果汽车被租赁，那么需要一个成员来描述其租赁公司。可以用下面的代码来完成：

```
struct owner {
    char socsecurity[12];
    ...
};
struct leasecompany {
    char name[40];
    char headquarters[40];
    ...
};

union data {
    struct owner owncar;
    struct leasecompany leasecar;
};
struct car_data {
    char make[15];
    int status; /* 私有为 0，租赁为 1 */
    union data ownerinfo;
    ...
};
```

假设 flits 是 car_data 类型的结构变量，如果 flits.status 为 0，程序将使用 flits.ownerinfo.owncar.socsecurity，如果 flits.status 为 1，程序则使用 flits.ownerinfo.leasecar.name。

14.10.2 匿名联合（C11）

匿名联合和匿名结构的工作原理相同，即匿名联合是一个结构或联合的无名联合成员。例如，我们重新定义 car_data 结构如下：

```
struct owner {
    char socsecurity[12];
    ...
};
struct leasecompany {
    char name[40];
    char headquarters[40];
    ...
```

```
};
struct car_data {
    char make[15];
    int status; /* 私有为 0，租赁为 1 */
    union {
        struct owner owncar;
        struct leasecompany leasecar;
    };
    ...
};
```

现在，如果 flits 是 car_data 类型的结构变量，可以用 flits.owncar.socsecurity 代替 flits.ownerinfo.owncar.socsecurity。

总结：结构和联合运算符

成员运算符：.

一般注释：

该运算符与结构或联合名一起使用，指定结构或联合的一个成员。如果 name 是一个结构的名称，member 是该结构模版指定的一个成员名，下面标识了该结构的这个成员：

```
name.member
```

name.member 的类型就是 member 的类型。联合使用成员运算符的方式与结构相同。

示例：

```
struct {
    int code;
    float cost;
} item;
item.code = 1265;
```

间接成员运算符：->

一般注释：

该运算符和指向结构或联合的指针一起使用，标识结构或联合的一个成员。假设 ptrstr 是指向结构的指针，member 是该结构模版指定的一个成员，那么：

```
ptrstr->member
```

标识了指向结构的成员。联合使用间接成员运算符的方式与结构相同。

示例：

```
struct {
    int code;
    float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;
```

最后一条语句把一个 int 类型的值赋给 item 的 code 成员。如下 3 个表达式是等价的：

```
ptrst->code        item.code         (*ptrst).code
```

14.11　枚举类型

可以用**枚举类型**（*enumerated type*）声明符号名称来表示整型常量。使用 enum 关键字，可以创建一个新“类型”并指定它可具有的值（实际上，enum 常量是 int 类型，因此，只要能使用 int 类型的地方就

可以使用枚举类型）。枚举类型的目的是提高程序的可读性。它的语法与结构的语法相同。例如，可以这样声明：

```
enum spectrum {red, orange, yellow, green, blue, violet};
enum spectrum color;
```

第 1 个声明创建了 spetrum 作为标记名，允许把 enum spetrum 作为一个类型名使用。第 2 个声明使 color 作为该类型的变量。第 1 个声明中花括号内的标识符枚举了 spectrum 变量可能有的值。因此，color 可能的值是 red、orange、yellow 等。这些符号常量被称为**枚举符**（*enumerator*）。然后，便可这样用：

```
int c;
color = blue;
if (color == yellow)
    ...;
for (color = red; color <= violet; color++)
    ...;
```

虽然枚举符（如 red 和 blue）是 int 类型，但是枚举变量可以是任意整数类型，前提是该整数类型可以储存枚举常量。例如，spectrum 的枚举符范围是 0～5，所以编译器可以用 unsigned char 来表示 color 变量。

顺带一提，C 枚举的一些特性并不适用于 C++。例如，C 允许枚举变量使用++运算符，但是 C++标准不允许。所以，如果编写的代码将来会并入 C++程序，那么必须把上面例子中的 color 声明为 int 类型，才能 C 和 C++都兼容。

14.11.1　enum 常量

blue 和 red 到底是什么？从技术层面看，它们是 int 类型的常量。例如，假定有前面的枚举声明，可以这样写：

```
printf("red = %d, orange = %d\n", red, orange);
```

其输出如下：

```
red = 0, orange = 1
```

red 成为一个有名称的常量，代表整数 0。类似地，其他标识符都是有名称的常量，分别代表 1～5。只要是能使用整型常量的地方就可以使用枚举常量。例如，在声明数组时，可以用枚举常量表示数组的大小；在 switch 语句中，可以把枚举常量作为标签。

14.11.2　默认值

默认情况下，枚举列表中的常量都被赋予 0、1、2 等。因此，下面的声明中 nina 的值是 3：

```
enum kids {nippy, slats, skippy, nina, liz};
```

14.11.3　赋值

在枚举声明中，可以为枚举常量指定整数值：

```
enum levels {low = 100, medium = 500, high = 2000};
```

如果只给一个枚举常量赋值，没有对后面的枚举常量赋值，那么后面的常量会被赋予后续的值。例如，假设有如下的声明：

```
enum feline {cat, lynx = 10, puma, tiger};
```

那么，cat 的值是 0（默认），lynx、puma 和 tiger 的值分别是 10、11、12。

14.11.4　enum 的用法

枚举类型的目的是为了提高程序的可读性和可维护性。如果要处理颜色，使用 red 和 blue 比使用 0 和 1 更直观。注意，枚举类型只能在内部使用。如果要输入 color 中 orange 的值，只能输入 1，而不是单词 orange。或者，让程序先读入字符串"orange"，再将其转换为 orange 代表的值。

因为枚举类型是整数类型，所以可以在表达式中以使用整数变量的方式使用 enum 变量。它们用在 case 语句中很方便。

程序清单 14.15 演示了一个使用 enum 的小程序。该程序示例使用默认值的方案，把 red 的值设置为 0，使之成为指向字符串"red"的指针的索引。

程序清单 14.15　enum.c 程序

```
/* enum.c -- 使用枚举类型的值 */
#include <stdio.h>
#include <string.h>    // 提供 strcmp()、strchr()函数的原型
#include <stdbool.h>   // C99 特性
char * s_gets(char * st, int n);

enum spectrum { red, orange, yellow, green, blue, violet };
const char * colors [] = { "red", "orange", "yellow",
"green", "blue", "violet" };
#define LEN 30

int main(void)
{
    char choice[LEN];
    enum spectrum color;
    bool color_is_found = false;

    puts("Enter a color (empty line to quit):");
    while (s_gets(choice, LEN) != NULL && choice[0] != '\0')
    {
        for (color = red; color <= violet; color++)
        {
            if (strcmp(choice, colors[color]) == 0)
            {
                color_is_found = true;
                break;
            }
        }
        if (color_is_found)
            switch (color)
        {
            case red: puts("Roses are red.");
                break;
            case orange: puts("Poppies are orange.");
                break;
            case yellow: puts("Sunflowers are yellow.");
                break;
            case green: puts("Grass is green.");
                break;
            case blue: puts("Bluebells are blue.");
```

```
                    break;
                case violet: puts("Violets are violet.");
                    break;
            }
        else
            printf("I don't know about the color %s.\n", choice);
        color_is_found = false;
        puts("Next color, please (empty line to quit):");
    }
    puts("Goodbye!");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;        // 清理输入行
    }
    return ret_val;
}
```

当输入的字符串与 color 数组的成员指向的字符串相匹配时，for 循环结束。如果循环找到匹配的颜色，程序就用枚举变量的值与作为 case 标签的枚举常量匹配。下面是该程序的一个运行示例：

```
Enter a color (empty line to quit):
blue
Bluebells are blue.
Next color, please (empty line to quit):
orange
Poppies are orange.
Next color, please (empty line to quit):
purple
I don't know about the color purple.
Next color, please (empty line to quit):

Goodbye!
```

14.11.5　共享名称空间

C 语言使用**名称空间**（*namespace*）标识程序中的各部分，即通过名称来识别。作用域是名称空间概念的一部分：两个不同作用域的同名变量不冲突；两个相同作用域的同名变量冲突。名称空间是分类别的。在特定作用域中的结构标记、联合标记和枚举标记都共享相同的名称空间，该名称空间与普通变量使用的空间不同。这意味着在相同作用域中变量和标记的名称可以相同，不会引起冲突，但是不能在相同作用域

中声明两个同名标签或同名变量。例如，在 C 中，下面的代码不会产生冲突：

```
struct rect { double x; double y; };
int rect; // 在C中不会产生冲突
```

尽管如此，以两种不同的方式使用相同的标识符会造成混乱。另外，C++不允许这样做，因为它把标记名和变量名放在相同的名称空间中。

14.12　typedef 简介

typedef 工具是一个高级数据特性，利用 typedef 可以为某一类型自定义名称。这方面与#define 类似，但是两者有 3 处不同：

- 与#define 不同，typedef 创建的符号名只受限于类型，不能用于值。
- typedef 由编译器解释，不是预处理器。
- 在其受限范围内，typedef 比#define 更灵活。

下面介绍 typedef 的工作原理。假设要用 BYTE 表示 1 字节的数组。只需像定义个 char 类型变量一样定义 BYTE，然后在定义前面加上关键字 typedef 即可：

```
typedef unsigned char BYTE;
```

随后，便可使用 BYTE 来定义变量：

```
BYTE x, y[10], * z;
```

该定义的作用域取决于 typedef 定义所在的位置。如果定义在函数中，就具有局部作用域，受限于定义所在的函数。如果定义在函数外面，就具有文件作用域。

通常，typedef 定义中用大写字母表示被定义的名称，以提醒用户这个类型名实际上是一个符号缩写。当然，也可以用小写：

```
typedef unsigned char byte;
```

typedef 中使用的名称遵循变量的命名规则。

为现有类型创建一个名称，看上去真是多此一举，但是它有时的确很有用。在前面的示例中，用 BYTE 代替 unsigned char 表明你打算用 BYTE 类型的变量表示数字，而不是字符码。使用 typedef 还能提高程序的可移植性。例如，我们之前提到的 sizeof 运算符的返回类型：size_t 类型，以及 time()函数的返回类型：time_t 类型。C 标准规定 sizeof 和 time()返回整数类型，但是让实现来决定具体是什么整数类型。其原因是，C 标准委员会认为没有哪个类型对于所有的计算机平台都是最优选择。所以，标准委员会决定建立一个新的类型名（如，time_t），并让实现使用 typedef 来设置它的具体类型。以这样的方式，C 标准提供以下通用原型：

```
time_t time(time_t *);
```

time_t 在一个系统中是 unsigned long，在另一个系统中可以是 unsigned long long。只要包含 time.h 头文件，程序就能访问合适的定义，你也可以在代码中声明 time_t 类型的变量。

typedef 的一些特性与#define 的功能重合。例如：

```
#define BYTE unsigned char
```

这使预处理器用 BYTE 替换 unsigned char。但是也有#define 没有的功能：

```
typedef char * STRING;
```

没有 typedef 关键字，编译器将把 STRING 识别为一个指向 char 的指针变量。有了 typedef 关键字，编译器则把 STRING 解释成一个类型的标识符，该类型是指向 char 的指针。因此：

```
STRING name, sign;
```

相当于：

```
char * name, * sign;
```

但是，如果这样假设：

```
#define STRING char *
```

然后，下面的声明：

```
STRING name, sign;
```

将被翻译成：

```
char * name, sign;
```

这导致只有 name 才是指针。

还可以把 typedef 用于结构：

```
typedef struct complex {
    float real;
    float imag;
} COMPLEX;
```

然后便可使用 COMPLEX 类型代替 complex 结构来表示复数。使用 typedef 的第 1 个原因是：为经常出现的类型创建一个方便、易识别的类型名。例如，前面的例子中，许多人更倾向于使用 STRING 或与其等价的标记。

用 typedef 来命名一个结构类型时，可以省略该结构的标签：

```
typedef struct {double x; double y;} rect;
```

假设这样使用 typedef 定义的类型名：

```
rect r1 = {3.0, 6.0};
rect r2;
```

以上代码将被翻译成：

```
struct {double x; double y;} r1= {3.0, 6.0};
struct {double x; double y;} r2;
r2 = r1;
```

这两个结构在声明时都没有标记，它们的成员完全相同（成员名及其类型都匹配），C 认为这两个结构的类型相同，所以 r1 和 r2 间的赋值是有效操作。

使用 typedef 的第 2 个原因是：typedef 常用于给复杂的类型命名。例如，下面的声明：

```
typedef char (* FRPTC ()) [5];
```

把 FRPTC 声明为一个函数类型，该函数返回一个指针，该指针指向内含 5 个 char 类型元素的数组（参见下一节的讨论）。

使用 typedef 时要记住，typedef 并没有创建任何新类型，它只是为某个已存在的类型增加了一个方便使用的标签。以前面的 STRING 为例，这意味着我们创建的 STRING 类型变量可以作为实参传递给以指向 char 指针作为形参的函数。

通过结构、联合和 typedef，C 提供了有效处理数据的工具和处理可移植数据的工具。

14.13 其他复杂的声明

C 允许用户自定义数据形式。虽然我们常用的是一些简单的形式，但是根据需要有时还会用到一些复杂的形式。在一些复杂的声明中，常包含下面的符号，如表 14.1 所示：

表 14.1　声明时可使用的符号

符号	含义
*	表示一个指针
()	表示一个函数
[]	表示一个数组

下面是一些较复杂的声明示例：

```
int board[8][8];        // 声明一个内含 int 数组的数组
int ** ptr;             // 声明一个指向指针的指针，被指向的指针指向 int
int * risks[10];        // 声明一个内含 10 个元素的数组，每个元素都是一个指向 int 的指针
int (* rusks)[10];      // 声明一个指向数组的指针，该数组内含 10 个 int 类型的值
int * oof[3][4];        // 声明一个 3×4 的二维数组，每个元素都是指向 int 的指针
int (* uuf)[3][4];      // 声明一个指向 3×4 二维数组的指针，该数组中内含 int 类型值
int (* uof[3])[4];      // 声明一个内含 3 个指针元素的数组，其中每个指针都指向一个内含 4 个 int 类
                           型元素的数组
```

要看懂以上声明，关键要理解*、()和[]的优先级。记住下面几条规则。

1. 数组名后面的[]和函数名后面的()具有相同的优先级。它们比*（解引用运算符）的优先级高。因此下面声明的 risk 是一个指针数组，不是指向数组的指针：

```
int * risks[10];
```

2. []和()的优先级相同，由于都是从左往右结合，所以下面的声明中，在应用方括号之前，*先与 rusks 结合。因此 rusks 是一个指向数组的指针，该数组内含 10 个 int 类型的元素：

```
int (* rusks)[10];
```

3. []和()都是从左往右结合。因此下面声明的 goods 是一个由 12 个内含 50 个 int 类型值的数组组成的二维数组，不是一个有 50 个内含 12 个 int 类型值的数组组成的二维数组：

```
int goods[12][50];
```

把以上规则应用于下面的声明：

```
int * oof[3][4];
```

[3]比*的优先级高，由于从左往右结合，所以[3]先与 oof 结合。因此，oof 首先是一个内含 3 个元素的数组。然后再与[4]结合，所以 oof 的每个元素都是内含 4 个元素的数组。*说明这些元素都是指针。最后，int 表明了这 4 个元素都是指向 int 的指针。因此，这条声明要表达的是：foo 是一个内含 3 个元素的数组，其中每个元素是由 4 个指向 int 的指针组成的数组。简而言之，oof 是一个 3×4 的二维数组，每个元素都是指向 int 的指针。编译器要为 12 个指针预留存储空间。

现在来看下面的声明：

```
int (* uuf)[3][4];
```

圆括号使得*先与 uuf 结合，说明 uuf 是一个指针，所以 uuf 是一个指向 3×4 的 int 类型二维数组的指针。编译器要为一个指针预留存储空间。

根据这些规则，还可以声明：

```
char * fump(int);           // 返回字符指针的函数
char (* frump)(int);        // 指向函数的指针，该函数的返回类型为 char
char (* flump[3])(int);     // 内含 3 个指针的数组，每个指针都指向返回类型为 char 的函数
```

这 3 个函数都接受 int 类型的参数。

可以使用 typedef 建立一系列相关类型：

```
typedef int arr5[5];
typedef arr5 * p_arr5;
typedef p_arr5 arrp10[10];
arr5 togs;    // togs 是一个内含 5 个 int 类型值的数组
p_arr5 p2;    // p2 是一个指向数组的指针，该数组内含 5 个 int 类型的值
arrp10 ap;    // ap 是一个内含 10 个指针的数组，每个指针都指向一个内含 5 个 int 类型值的数组
```

如果把这些放入结构中，声明会更复杂。至于应用，我们就不再进一步讨论了。

14.14 函数和指针

通过上一节的学习可知，可以声明一个指向函数的指针。这个复杂的玩意儿到底有何用处？通常，函数指针常用作另一个函数的参数，告诉该函数要使用哪一个函数。例如，排序数组涉及比较两个元素，以确定先后。如果元素是数字，可以使用>运算符；如果元素是字符串或结构，就要调用函数进行比较。C 库中的 qsort()函数可以处理任意类型的数组，但是要告诉 qsort()使用哪个函数来比较元素。为此，qsort()函数的参数列表中，有一个参数接受指向函数的指针。然后，qsort()函数使用该函数提供的方案进行排序，无论这个数组中的元素是整数、字符串还是结构。

我们来进一步研究函数指针。首先，什么是函数指针？假设有一个指向 int 类型变量的指针，该指针储存着这个 int 类型变量储存在内存位置的地址。同样，函数也有地址，因为函数的机器语言实现由载入内存的代码组成。指向函数的指针中储存着函数代码的起始处的地址。

其次，声明一个数据指针时，必须声明指针所指向的数据类型。声明一个函数指针时，必须声明指针指向的函数类型。为了指明函数类型，要指明函数签名，即函数的返回类型和形参类型。例如，考虑下面的函数原型：

```
void ToUpper(char *);  // 把字符串中的字符转换成大写字符
```

ToUpper()函数的类型是“带 char * 类型参数、返回类型是 void 的函数”。下面声明了一个指针 pf 指向该函数类型：

```
void (*pf)(char *);    // pf 是一个指向函数的指针
```

从该声明可以看出，第 1 对圆括号把*和 pf 括起来，表明 pf 是一个指向函数的指针。因此，(*pf)是一个参数列表为(char *)、返回类型为 void 的函数。注意，把函数名 ToUpper 替换为表达式(*pf)是创建指向函数指针最简单的方式。所以，如果想声明一个指向某类型函数的指针，可以写出该函数的原型后把函数名替换成(*pf)形式的表达式，创建函数指针声明。前面提到过，由于运算符优先级的规则，在声明函数指针时必须把*和指针名括起来。如果省略第 1 个圆括号会导致完全不同的情况：

```
void *pf(char *); // pf 是一个返回字符指针的函数
```

> **提示**
>
> 要声明一个指向特定类型函数的指针，可以先声明一个该类型的函数，然后把函数名替换成(*pf)形式的表达式。然后，pf 就成为指向该类型函数的指针。

声明了函数指针后，可以把类型匹配的函数地址赋给它。在这种上下文中，函数名可以用于表示函数的地址：

```
void ToUpper(char *);
```

```
void ToLower(char *);
int round(double);
void (*pf)(char *);
pf = ToUpper;      // 有效，ToUpper 是该类型函数的地址
pf = ToLower;      //有效，ToUpper 是该类型函数的地址
pf = round;        // 无效，round 与指针类型不匹配
pf = ToLower();    // 无效，ToLower()不是地址
```

最后一条语句是无效的，不仅因为 ToLower()不是地址，而且 ToLower()的返回类型是 void，它没有返回值，不能在赋值语句中进行赋值。注意，指针 pf 可以指向其他带 char *类型参数、返回类型是 void 的函数，不能指向其他类型的函数。

既然可以用数据指针访问数据，也可以用函数指针访问函数。奇怪的是，有两种逻辑上不一致的语法可以这样做，下面解释：

```
void ToUpper(char *);
void ToLower(char *);
void (*pf)(char *);
char mis[] = "Nina Metier";
pf = ToUpper;
(*pf)(mis);   // 把 ToUpper 作用于（语法 1）
pf = ToLower;
pf(mis);      // 把 ToLower 作用于（语法 2）
```

这两种方法看上去都合情合理。先分析第 1 种方法：由于 pf 指向 ToUpper 函数，那么*pf 就相当于 ToUpper 函数，所以表达式(*pf)(mis)和 ToUpper(mis)相同。从 ToUpper 函数和 pf 的声明就能看出，ToUpper 和(*pf)是等价的。第 2 种方法：由于函数名是指针，那么指针和函数名可以互换使用，所以 pf(mis)和 ToUpper(mis)相同。从 pf 的赋值表达式语句就能看出 ToUpper 和 pf 是等价的。由于历史的原因，贝尔实验室的 C 和 UNIX 的开发者采用第 1 种形式，而伯克利的 UNIX 推广者却采用第 2 种形式。K&R C 不允许第 2 种形式。但是，为了与现有代码兼容，ANSI C 认为这两种形式（本例中是(*pf)(mis)和 pf(mis)）等价。后续的标准也延续了这种矛盾的和谐。

作为函数的参数是数据指针最常见的用法之一，函数指针亦如此。例如，考虑下面的函数原型：

```
void show(void (* fp)(char *), char * str);
```

这看上去让人头晕。它声明了两个形参：fp 和 str。fp 形参是一个函数指针，str 是一个数据指针。更具体地说，fp 指向的函数接受 char * 类型的参数，其返回类型为 void；str 指向一个 char 类型的值。因此，假设有上面的声明，可以这样调用函数：

```
show(ToLower, mis);    /* show()使用 ToLower()函数：fp = ToLower */
show(pf, mis);         /* show()使用 pf 指向的函数： fp = pf */
```

show()如何使用传入的函数指针？是用 fp()语法还是(*fp)()语法调用函数：

```
void show(void (* fp)(char *), char * str)
{
     (*fp)(str);  /* 把所选函数作用于 str */
    puts(str);         /* 显示结果 */
}
```

例如，这里的 show()首先用 fp 指向的函数转换 str，然后显示转换后的字符串。

顺带一提，把带返回值的函数作为参数传递给另一个函数有两种不同的方法。例如，考虑下面的语句：

```
function1(sqrt);        /* 传递 sqrt()函数的地址 */
function2(sqrt(4.0));  /* 传递 sqrt()函数的返回值 */
```

第 1 条语句传递的是 sqrt()函数的地址，假设 function1()在其代码中会使用该函数。第 2 条语句先调用 sqrt()函数，然后求值，并把返回值（该例中是 2.0）传递给 function2()。

程序清单 14.16 中的程序通过 show()函数来演示这些要点，该函数以各种转换函数作为参数。该程序也演示了一些处理菜单的有用技巧。

程序清单 14.16　func_ptr.c 程序

```
// func_ptr.c -- 使用函数指针
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define LEN 81
char * s_gets(char * st, int n);
char showmenu(void);
void eatline(void);         // 读取至行末尾
void show(void(*fp)(char *), char * str);
void ToUpper(char *);       // 把字符串转换为大写
void ToLower(char *);       // 把字符串转换为小写
void Transpose(char *);     // 大小写转置
void Dummy(char *);         // 不更改字符串

int main(void)
{
    char line[LEN];
    char copy[LEN];
    char choice;
    void(*pfun)(char *); // 声明一个函数指针，被指向的函数接受char *类型的参数，无返回值

    puts("Enter a string (empty line to quit):");
    while (s_gets(line, LEN) != NULL && line[0] != '\0')
    {
        while ((choice = showmenu()) != 'n')
        {
            switch (choice)  // switch语句设置指针
            {
            case 'u': pfun = ToUpper;   break;
            case 'l': pfun = ToLower;   break;
            case 't': pfun = Transpose; break;
            case 'o': pfun = Dummy;     break;
            }
            strcpy(copy, line);     // 为show()函数拷贝一份
            show(pfun, copy);       // 根据用户的选择，使用选定的函数
        }
        puts("Enter a string (empty line to quit):");
    }
    puts("Bye!");

    return 0;
}

char showmenu(void)
{
    char ans;
```

```
    puts("Enter menu choice:");
    puts("u) uppercase      l) lowercase");
    puts("t) transposed case o) original case");
    puts("n) next string");
    ans = getchar();       // 获取用户的输入
    ans = tolower(ans);    // 转换为小写
    eatline();             // 清理输入行
    while (strchr("ulton", ans) == NULL)
    {
        puts("Please enter a u, l, t, o, or n:");
        ans = tolower(getchar());
        eatline();
    }

    return ans;
}

void eatline(void)
{
    while (getchar() != '\n')
        continue;
}

void ToUpper(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}

void ToLower(char * str)
{
    while (*str)
    {
        *str = tolower(*str);
        str++;
    }
}
void Transpose(char * str)
{
    while (*str)
    {
        if (islower(*str))
            *str = toupper(*str);
        else if (isupper(*str))
            *str = tolower(*str);
        str++;
    }
}

void Dummy(char * str)
{
```

```
    // 不改变字符串
}

void show(void(*fp)(char *), char * str)
{
    (*fp)(str);   // 把用户选定的函数作用于str
    puts(str);    // 显示结果
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;          // 清理输入行中剩余的字符
    }
    return ret_val;
}
```

下面是该程序的输出示例：

```
Enter a string (empty line to quit):
Does C make you feel loopy?
Enter menu choice:
u) uppercase l) lowercase
t) transposed case o) original case
n) next string
t
dOES c MAKE YOU FEEL LOOPY?
Enter menu choice:
u) uppercase l) lowercase
t) transposed case o) original case
n) next string
l
does c make you feel loopy?
Enter menu choice:
u) uppercase l) lowercase
t) transposed case o) original case
n) next string
n
Enter a string (empty line to quit):

Bye!
```

注意，`ToUpper()`、`ToLower()`、`Transpose()`和`Dummy()`函数的类型都相同，所以这4个函数都可以赋给`pfun`指针。该程序把`pfun`作为`show()`的参数，但是也可以直接把这4个函数中的任一个函

数名作为参数，如 show(Transpose, copy)。

这种情况下，可以使用 typedef。例如，该程序中可以这样写：

```
typedef void (*V_FP_CHARP)(char *);
void show (V_FP_CHARP fp, char *);
V_FP_CHARP pfun;
```

如果还想更复杂一些，可以声明并初始化一个函数指针的数组：

```
V_FP_CHARP arpf[4] = {ToUpper, ToLower, Transpose, Dummy};
```

然后把 showmenu()函数的返回类型改为 int，如果用户输入 u，则返回 0；如果用户输入 l，则返回 2；如果用户输入 t，则返回 2，以此类推。可以把程序中的 switch 语句替换成下面的 while 循环：

```
index = showmenu();
while (index >= 0 && index <= 3)
{
    strcpy(copy, line);          /* 为 show()拷贝一份 */
    show(arpf[index], copy);     /* 使用选定的函数 */
    index = showmenu();
}
```

虽然没有函数数组，但是可以有函数指针数组。

以上介绍了使用函数名的 4 种方法：定义函数、声明函数、调用函数和作为指针。图 14.4 进行了总结。

```
                               函数原型中的函数名:  int comp(int x, int y);
函数调用中的函数名: 函数定义中的函数名:  status = comp(q,r);
                                                   int comp(intx, inty)
                                                   { ...
在赋值表达式语句中作为指针的函数名:  pfunct = comp;
                      作为指针参数的函数名:  slowsort(arr,n,comp);
```

图 14.4　函数名的用法

至于如何处理菜单，showmenu()函数给出了几种技巧。首先，下面的代码：

```
ans = getchar();        // 获取用户输入
ans = tolower(ans);     // 转换成小写
```

和

```
ans = tolower(getchar());
```

演示了转换用户输入的两种方法。这两种方法都可以把用户输入的字符转换为一种大小写形式，这样就不用检测用户输入的是'u'还是'U'，等等。

eatline()函数丢弃输入行中的剩余字符，在处理这两种情况时很有用。第一，用户为了输入一个选择，输入一个字符，然后按下 Enter 键，将产生一个换行符。如果不处理这个换行符，它将成为下一次读取的第 1 个字符。第二，假设用户输入的是整个单词 uppercase，而不是一个字母 u。如果 没有 eatline()函数，程序会把 uppercase 中的字符作为用户的响应依次读取。有了 eatline()，程序会读取 u 字符并丢弃输入行中剩余的字符。

其次，showmenu()函数的设计意图是，只给程序返回正确的选项。为完成这项任务，程序使用了 string.h 头文件中的标准库函数 strchr()：

```
while (strchr("ulton", ans) == NULL)
```

该函数在字符串"ulton"中查找字符 ans 首次出现的位置，并返回一个指向该字符的指针。如果没有找到该字符，则返回空指针。因此，上面的 while 循环头可以用下面的 while 循环头代替，但是上面的

用起来更方便：

```
while (ans != 'u' && ans != 'l' && ans != 't' && ans != 'o' && ans != 'n')
```

待检查的项越多，使用 strchr()就越方便。

14.15　关键概念

我们在编程中要表示的信息通常不只是一个数字或一些列数字。程序可能要处理具有多种属性的实体。例如，通过姓名、地址、电话号码和其他信息表示一名客户；或者，通过电影名、发行人、播放时长、售价等表示一部电影 DVD。C 结构可以把这些信息都放在一个单元内。在组织程序时这很重要，因为这样可以把相关的信息都储存在一处，而不是分散储存在多个变量中。

设计结构时，开发一个与之配套的函数包通常很有用。例如，写一个以结构（或结构的地址）为参数的函数打印结构内容，比用一堆 printf()语句强得多。因为只需要一个参数就能打印结构中的所有信息。如果把信息放到零散的变量中，每个部分都需要一个参数。另外，如果要在结构中增加一个成员，只需重写函数，不必改写函数调用。这在修改结构时很方便。

联合声明与结构声明类似。但是，联合的成员共享相同的存储空间，而且在联合中同一时间内只能有一个成员。实质上，可以在联合变量中储存一个类型不唯一的值。

enum 工具提供一种定义符号常量的方法，typedef 工具提供一种为基本或派生类型创建新标识符的方法。

指向函数的指针提供一种告诉函数应使用哪一个函数的方法。

14.16　本章小结

C 结构提供在相同的数据对象中储存多个不同类型数据项的方法。可以使用标记来标识一个具体的结构模板，并声明该类型的变量。通过成员点运算符（.）可以使用结构模版中的标签来访问结构的各个成员。

如果有一个指向结构的指针，可以用该指针和间接成员运算符（->）代替结构名和点运算符来访问结构的各成员。和数组不同，结构名不是结构的地址，要在结构名前使用&运算符才能获得结构的地址。

一贯以来，与结构相关的函数都使用指向结构的指针作为参数。现在的 C 允许把结构作为参数传递，作为返回值和同类型结构之间赋值。然而，传递结构的地址通常更有效。

联合使用与结构相同的语法。然而，联合的成员共享一个共同的存储空间。联合同一时间内只能储存一个单独的数据项，不像结构那样同时储存多种数据类型。也就是说，结构可以同时储存一个 int 类型数据、一个 double 类型数据和一个 char 类型数据，而相应的联合只能保存一个 int 类型数据，或者一个 double 类型数据，或者一个 char 类型数据。

通过枚举可以创建一系列代表整型常量（枚举常量）的符号和定义相关联的枚举类型。

typedef 工具可用于建立 C 标准类型的别名或缩写。

函数名代表函数的地址，可以把函数的地址作为参数传递给其他函数，然后这些函数就可以使用被指向的函数。如果把特定函数的地址赋给一个名为 pf 的函数指针，可以通过以下两种方式调用该函数：

```
#include <math.h> /* 提供 sin()函数的原型：double sin(double) */
...
double (*pdf)(double);
double x;
pdf = sin;
```

```
x = (*pdf)(1.2);   // 调用 sin(1.2)
x = pdf(1.2);      // 同样调用 sin(1.2)
```

14.17　复习题

复习题的参考答案在附录 A 中。

1. 下面的结构模板有什么问题：

```
structure {
    char itable;
    int num[20];
    char * togs
}
```

2. 下面是程序的一部分，输出是什么？

```
#include <stdio.h>
struct house {
    float sqft;
    int rooms;
    int stories;
    char address[40];
};
int main(void)
{
    struct house fruzt = {1560.0, 6, 1, "22 Spiffo Road"};
    struct house *sign;

    sign = &fruzt;
    printf("%d %d\n", fruzt.rooms, sign->stories);
    printf("%s \n", fruzt.address);
    printf("%c %c\n", sign->address[3], fruzt.address[4]);
    return 0;
}
```

3. 设计一个结构模板储存一个月份名、该月份名的 3 个字母缩写、该月的天数以及月份号。

4. 定义一个数组，内含 12 个结构（第 3 题的结构类型）并初始化为一个年份（非闰年）。

5. 编写一个函数，用户提供月份号，该函数就返回一年中到该月为止（包括该月）的总天数。假设在所有函数的外部声明了第 3 题的结构模版和一个该类型结构的数组。

6. a．假设有下面的 `typedef`，声明一个内含 10 个指定结构的数组。然后，单独给成员赋值（或等价字符串），使第 3 个元素表示一个焦距长度有 `500mm`，孔径为 `f/2.0` 的 Remarkata 镜头。

```
typedef struct lens {    /* 描述镜头              */
    float foclen;        /* 焦距长度，单位为 mm   */
    float fstop;         /* 孔径                  */
    char brand[30];      /* 品牌名称              */
} LENS;
```

   b．重写 a，在声明中使用一个待指定初始化器的初始化列表，而不是对每个成员单独赋值。

7. 考虑下面程序片段：

```
struct name {
    char first[20];
    char last[20];
};
```

```
struct bem {
    int limbs;
    struct name title;
    char type[30];
};
struct bem * pb;
struct bem deb = {
    6,
    { "Berbnazel", "Gwolkapwolk" },
    "Arcturan"
};

pb = &deb;
```

a. 下面的语句分别打印什么？

```
printf("%d\n", deb.limbs);
printf("%s\n", pb->type);
printf("%s\n", pb->type + 2);
```

b. 如何用结构表示法（两种方法）表示"Gwolkapwolk"？

c. 编写一个函数，以 bem 结构的地址作为参数，并以下面的形式输出结构的内容（假定结构模板在一个名为 starfolk.h 的头文件中）：

```
Berbnazel Gwolkapwolk is a 6-limbed Arcturan.
```

8. 考虑下面的声明：

```
struct fullname {
    char fname[20];
    char lname[20];
};
struct bard {
    struct fullname name;
    int born;
    int died;
};
struct bard willie;
struct bard *pt = &willie;
```

a. 用 willie 标识符标识 willie 结构的 born 成员。

b. 用 pt 标识符标识 willie 结构的 born 成员。

c. 调用 scanf()读入一个用 willie 标识符标识的 born 成员的值。

d. 调用 scanf()读入一个用 pt 标识符标识的 born 成员的值。

e. 调用 scanf()读入一个用 willie 标识符标识的 name 成员中 lname 成员的值。

f. 调用 scanf()读入一个用 pt 标识符标识的 name 成员中 lname 成员的值。

g. 构造一个标识符，标识 willie 结构变量所表示的姓名中名的第 3 个字母（英文的名在前）。

h. 构造一个表达式，表示 willie 结构变量所表示的名和姓中的字母总数。

9. 定义一个结构模板以储存这些项：汽车名、马力、EPA（美国环保局）城市交通 MPG（每加仑燃料行驶的英里数）评级、轴距和出厂年份。使用 car 作为该模版的标记。

10. 假设有如下结构：

```
struct gas {
    float distance;
    float gals;
```

```
    float mpg;
};
```

a. 设计一个函数，接受 struct gas 类型的参数。假设传入的结构包含 distance 和 gals 信息。该函数为 mpg 成员计算正确的值，并把值返回该结构。

b. 设计一个函数，接受 struct gas 类型的参数。假设传入的结构包含 distance 和 gals 信息。该函数为 mpg 成员计算正确的值，并把该值赋给合适的成员。

11. 声明一个标记为 choices 的枚举，把枚举常量 no、yes 和 maybe 分别设置为 0、1、2。

12. 声明一个指向函数的指针，该函数返回指向 char 的指针，接受一个指向 char 的指针和一个 char 类型的值。

13. 声明 4 个函数，并初始化一个指向这些函数的指针数组。每个函数都接受两个 double 类型的参数，返回 double 类型的值。另外，用两种方法使用该数组调用带 10.0 和 2.5 实参的第 2 个函数。

14.18　编程练习

1. 重新编写复习题 5，用月份名的拼写代替月份号（别忘了使用 strcmp()）。在一个简单的程序中测试该函数。

2. 编写一个函数，提示用户输入日、月和年。月份可以是月份号、月份名或月份名缩写。然后该程序应返回一年中到用户指定日子（包括这一天）的总天数。

3. 修改程序清单 14.2 中的图书目录程序，使其按照输入图书的顺序输出图书的信息，然后按照标题字母的声明输出图书的信息，最后按照价格的升序输出图书的信息。

4. 编写一个程序，创建一个有两个成员的结构模板：

   a. 第 1 个成员是社会保险号，第 2 个成员是一个有 3 个成员的结构，第 1 个成员代表名，第 2 个成员代表中间名，第 3 个成员表示姓。创建并初始化一个内含 5 个该类型结构的数组。该程序以下面的格式打印数据：

   ```
   Dribble, Flossie M. –– 302039823
   ```

   如果有中间名，只打印它的第 1 个字母，后面加一个点（.）；如果没有中间名，则不用打印点。编写一个程序进行打印，把结构数组传递给这个函数。

   b. 修改 a 部分，传递结构的值而不是结构的地址。

5. 编写一个程序满足下面的要求。

   a. 外部定义一个有两个成员的结构模板 name：一个字符串储存名，一个字符串储存姓。

   b. 外部定义一个有 3 个成员的结构模板 student：一个 name 类型的结构，一个 grade 数组储存 3 个浮点型分数，一个变量储存 3 个分数平均数。

   c. 在 main()函数中声明一个内含 CSIZE（CSIZE = 4）个 student 类型结构的数组，并初始化这些结构的名字部分。用函数执行 g、e、f 和 g 中描述的任务。

   d. 以交互的方式获取每个学生的成绩，提示用户输入学生的姓名和分数。把分数储存到 grade 数组相应的结构中。可以在 main()函数或其他函数中用循环来完成。

   e. 计算每个结构的平均分，并把计算后的值赋给合适的成员。

   f. 打印每个结构的信息。

g. 打印班级的平均分，即所有结构的数值成员的平均值。

6. 一个文本文件中保存着一个垒球队的信息。每行数据都是这样排列：

```
4 Jessie Joybat 5 2 1 1
```

第 1 项是球员号，为方便起见，其范围是 0～18。第 2 项是球员的名。第 3 项是球员的姓。名和姓都是一个单词。第 4 项是官方统计的球员上场次数。接着 3 项分别是击中数、走垒数和打点（RBI）。文件可能包含多场比赛的数据，所以同一位球员可能有多行数据，而且同一位球员的多行数据之间可能有其他球员的数据。编写一个程序，把数据储存到一个结构数组中。该结构中的成员要分别表示球员的名、姓、上场次数、击中数、走垒数、打点和安打率（稍后计算）。可以使用球员号作为数组的索引。该程序要读到文件结尾，并统计每位球员的各项累计总和。

世界棒球统计与之相关。例如，一次走垒和触垒中的失误不计入上场次数，但是可能产生一个 RBI。但是该程序要做的是像下面描述的一样读取和处理数据文件，不会关心数据的实际含义。

要实现这些功能，最简单的方法是把结构的内容都初始化为零，把文件中的数据读入临时变量中，然后将其加入相应的结构中。程序读完文件后，应计算每位球员的安打率，并把计算结果储存到结构的相应成员中。计算安打率是用球员的累计击中数除以上场累计次数。这是一个浮点数计算。最后，程序结合整个球队的统计数据，一行显示一位球员的累计数据。

7. 修改程序清单 14.14，从文件中读取每条记录并显示出来，允许用户删除记录或修改记录的内容。如果删除记录，把空出来的空间留给下一个要读入的记录。要修改现有的文件内容，必须用"r+b"模式，而不是"a+b"模式。而且，必须更加注意定位文件指针，防止新加入的记录覆盖现有记录。最简单的方法是改动储存在内存中的所有数据，然后再把最后的信息写入文件。跟踪的一个方法是在 book 结构中添加一个成员表示是否该项被删除。

8. 巨人航空公司的机群由 12 个座位的飞机组成。它每天飞行一个航班。根据下面的要求，编写一个座位预订程序。

   a. 该程序使用一个内含 12 个结构的数组。每个结构中包括：一个成员表示座位编号、一个成员表示座位是否已被预订、一个成员表示预订人的名、一个成员表示预订人的姓。

   b. 该程序显示下面的菜单：

   ```
   To choose a function, enter its letter label:
   a) Show number of empty seats
   b) Show list of empty seats
   c) Show alphabetical list of seats
   d) Assign a customer to a seat assignment
   e) Delete a seat assignment
   f) Quit
   ```

   c. 该程序能成功执行上面给出的菜单。选择 d)和 e)要提示用户进行额外输入，每个选项都能让用户中止输入。

   d. 执行特定程序后，该程序再次显示菜单，除非用户选择 f)。

9. 巨人航空公司（编程练习 8）需要另一架飞机（容量相同），每天飞 4 班（航班 102、311、444 和 519）。把程序扩展为可以处理 4 个航班。用一个顶层菜单提供航班选择和退出。选择一个特定航班，就会出现和编程练习 8 类似的菜单。但是该菜单要添加一个新选项：确认座位分配。而且，菜单中的退出是返回顶层菜单。每次显示都要指明当前正在处理的航班号。另外，座位分配显示要指明确认状态。

10. 编写一个程序，通过一个函数指针数组实现菜单。例如，选择菜单中的 a，将激活由该数组第 1

个元素指向的函数。

11. 编写一个名为 `transform()`的函数，接受 4 个参数：内含 `double` 类型数据的源数组名、内含 `double` 类型数据的目标数组名、一个表示数组元素个数的 `int` 类型参数、函数名（或等价的函数指针）。`transform()`函数应把指定函数应用于源数组中的每个元素，并把返回值储存在目标数组中。例如：

```
transform(source, target, 100, sin);
```

该声明会把 `target[0]`设置为 `sin(source[0])`，等等，共有 100 个元素。在一个程序中调用 `transform()`4 次，以测试该函数。分别使用 `math.h` 函数库中的两个函数以及自定义的两个函数作为参数。

第 15 章 位操作

本章介绍以下内容：

■ 运算符：~、&、|、^、
<<、>>
&=、|=、^=、>>=、<<=

■ 二进制、十进制和十六进制记数法（复习）

■ 处理一个值中的位的两个 C 工具：位运算符和位字段

■ 关键字：_Alignas、_Alignof

在 C 语言中，可以单独操控变量中的位。读者可能好奇，竟然有人想这样做。有时必须单独操控位，而且非常有用。例如，通常向硬件设备发送一两个字节来控制这些设备，其中每个位（*bit*）都有特定的含义。另外，与文件相关的操作系统信息经常被储存，通过使用特定位表明特定项。许多压缩和加密操作都是直接处理单独的位。高级语言一般不会处理这级别的细节，C 在提供高级语言便利的同时，还能在为汇编语言所保留的级别上工作，这使其成为编写设备驱动程序和嵌入式代码的首选语言。

首先要介绍位、字节、二进制记数法和其他进制记数系统的一些背景知识。

15.1 二进制数、位和字节

通常都是基于数字 10 来书写数字。例如 2157 的千位是 2，百位是 1，十位是 5，个位是 7，可以写成：

$2\times1000 + 1\times100 + 5\times10 + 7\times1$

注意，1000 是 10 的立方（即 3 次幂），100 是 10 的平方（即 2 次幂），10 是 10 的 1 次幂，而且 10（以及任意正数）的 0 次幂是 1。因此，2157 也可以写成：

$2\times10^3+ 1\times10^2+ 5\times10^1+ 7\times10^0$

因为这种书写数字的方法是基于 10 的幂，所以称以 10 为基底书写 2157。

姑且认为十进制系统得以发展是得益于我们都有 10 根手指。从某种意义上看，计算机的位只有 2 根手指，因为它只能被设置为 0 或 1，关闭或打开。因此，计算机适用基底为 2 的数制系统。它用 2 的幂而不是 10 的幂。以 2 为基底表示的数字被称为**二进制数**（*binary number*）。二进制中的 2 和十进制中的 10 作用相同。例如，二进制数 1101 可表示为：

$1\times2^3+ 1\times2^2+ 0\times2^1+ 1\times2^0$

以十进制数表示为：

$1\times8 + 1\times4 + 0\times2 + 1\times1 = 13$

用二进制系统可以把任意整数（如果有足够的位）表示为 0 和 1 的组合。由于数字计算机通过关闭和打开状态的组合来表示信息，这两种状态分别用 0 和 1 来表示，所以使用这套数制系统非常方便。接下来，我们来学习二进制系统如何表示 1 字节的整数。

15.1.1　二进制整数

通常，1 字节包含 8 位。C 语言用**字节**（*byte*）表示储存系统字符集所需的大小，所以 C 字节可能是 8 位、9 位、16 位或其他值。不过，描述存储器芯片和数据传输率中所用的字节指的是 8 位字节。为了简化起见，本章假设 1 字节是 8 位（计算机界通常用**八位组（*octet*）**这个术语特指 8 位字节）。可以从左往右给这 8 位分别编号为 7～0。在 1 字节中，编号是 7 的位被称为**高阶位**（*high-order bit*），编号是 0 的位被称为**低阶位**（*low-order bit*）。每 1 位的编号对应 2 的相应指数。因此，可以根据图 15.1 所示的例子理解字节。



图 15.1　位编号和位值

这里，128 是 2 的 7 次幂，以此类推。该字节能表示的最大数字是把所有位都设置为 1：11111111。这个二进制数的值是：

```
128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255
```

而该字节最小的二进制数是 00000000，其值为 0。因此，1 字节可储存 0～255 范围内的数字，总共 256 个值。或者，通过不同的方式解释**位组合**（*bit pattern*），程序可以用 1 字节储存-128～+127 范围内的整数，总共还是 256 个值。例如，通常 `unsigned char` 用 1 字节表示的范围是 0～255，而 `signed char` 用 1 字节表示的范围是-128～+127。

15.1.2　有符号整数

如何表示有符号整数取决于硬件，而不是 C 语言。也许表示有符号数最简单的方式是用 1 位（如，高阶位）储存符号，只剩下 7 位表示数字本身（假设储存在 1 字节中）。用这种**符号量**（*sign-magnitude*）表示法，10000001 表示−1，00000001 表示 1。因此，其表示范围是−127～+127。

这种方法的缺点是有两个 0：+0 和-0。这很容易混淆，而且用两个位组合来表示一个值也有些浪费。

二进制补码（*two's-complement*）方法避免了这个问题，是当今最常用的系统。我们将以 1 字节为例，讨论这种方法。二进制补码用 1 字节中的后 7 位表示 0～127，高阶位设置为 0。目前，这种方法和符号量的方法相同。另外，如果高阶位是 1，表示的值为负。这两种方法的区别在于如何确定负值。从一个 9 位组合 100000000（256 的二进制形式）减去一个负数的位组合，结果是该负值的量。例如，假设一个负值的位组合是 10000000，作为一个无符号字节，该组合为表示 128；作为一个有符号值，该组合表示负值（编码是 7 的位为 1），而且值为 100000000-10000000，即 1000000（128）。因此，该数是-128（在符号量表示法中，该位组合表示−0）。类似地，10000001 是−127，11111111 是−1。该方法可以表示−128～+127 范围内的数。

要得到一个二进制补码数的相反数，最简单的方法是反转每一位（即 0 变为 1，1 变为 0），然后加 1。因为 1 是 00000001，那么−1 则是 11111110+1，或 11111111。这与上面的介绍一致。

二进制反码（*one's-complement*）方法通过反转位组合中的每一位形成一个负数。例如，00000001 是 1，那么 11111110 是−1。这种方法也有一个−0：11111111。该方法能表示-127～+127 之间的数。

15.1.3 二进制浮点数

浮点数分两部分储存：二进制小数和二进制指数。下面我们将详细介绍。

1. 二进制小数

一个普通的浮点数 0.527，表示如下：

```
5/10 + 2/100 + 7/1000
```

从左往右，各分母都是 10 的递增次幂。在二进制小数中，使用 2 的幂作为分母，所以二进制小数.101 表示为：

```
1/2 + 0/4 + 1/8
```

用十进制表示法为：

```
0.50 + 0.00 + 0.125
```

即是 0.625。

许多分数（如，1/3）不能用十进制表示法精确地表示。与此类似，许多分数也不能用二进制表示法准确地表示。实际上，二进制表示法只能精确地表示多个 1/2 的幂的和。因此，3/4 和 7/8 可以精确地表示为二进制小数，但是 1/3 和 2/5 却不能。

2. 浮点数表示法

为了在计算机中表示一个浮点数，要留出若干位（因系统而异）储存二进制分数，其他位储存指数。一般而言，数字的实际值是由二进制小数乘以 2 的指定次幂组成。例如，一个浮点数乘以 4，那么二进制小数不变，其指数乘以 2，二进制分数不变。如果一份浮点数乘以一个不是 2 的幂的数，会改变二进制小数部分，如有必要，也会改变指数部分。

15.2 其他进制数

计算机界通常使用八进制记数系统和十六进制记数系统。因为 8 和 16 都是 2 的幂，这些系统比十进制系统更接近计算机的二进制系统。

15.2.1 八进制

八进制（*octal*）是指八进制记数系统。该系统基于 8 的幂，用 0～7 表示数字（正如十进制用 0～9 表示数字一样）。例如，八进制数 451（在 C 中写作 0451）表示为：

$4\times8^2 + 5\times8^1 + 1\times8^0 = 297$（十进制）

了解八进制的一个简单的方法是，每个八进制位对应 3 个二进制位。表 15.1 列出了这种对应关系。这种关系使得八进制与二进制之间的转换很容易。例如，八进制数 0377 的二进制形式是 11111111。即，用 111 代替 0377 中的最后一个 7，再用 111 代替倒数第 2 个 7，最后用 011 代替 3，并舍去第 1 位的 0。这表明比 0377 大的八进制要用多个字节表示。这是八进制唯一不方便的地方：一个 3 位的八进制数可能要用 9 位二进制数来表示。注意，将八进制数转换为二进制形式时，不能去掉中间的 0。例如，八进制数 0173 的二进制形式是 01111011，不是 0111111。

表 15.1　与八进制位等价的二进制位

八进制位	等价的二进制位	八进制位	等价的二进制位
0	000	4	100
1	001	5	101
2	010	6	110
3	011	7	111

15.2.2　十六进制

十六进制（*hexadecimal* 或 *hex*）是指十六进制记数系统。该系统基于 16 的幂，用 0～15 表示数字。但是，由于没有单独的数（*digit*，即 0～9 这样单独一位的数）表示 10～15，所以用字母 A～F 来表示。例如，十六进制数 A3F（在 C 中写作 0xA3F）表示为：

$10\times16^2+3\times16^1+15\times16^0=2623$（十进制）

由于 A 表示 10，F 表示 15。在 C 语言中，A～F 既可用小写也可用大写。因此，2623 也可写作 0xa3f。

每个十六进制位都对应一个 4 位的二进制数（即 4 个二进制位），那么两个十六进制位恰好对应一个 8 位字节。第 1 个十六进制表示前 4 位，第 2 个十六进制位表示后 4 位。因此，十六进制很适合表示字节值。

表 15.2 列出了各进制之间的对应关系。例如，十六进制值 0xC2 可转换为 11000010。相反，二进制值 11010101 可以看作是 1101 0101，可转换为 0xD5。

表 15.2　十进制、十六进制和等价的二进制

十进制	十六进制	等价二进制	十进制	十六进制	等价二进制
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	10	A	1010
3	3	0011	11	B	1011
4	4	0100	12	C	1100
5	5	0101	13	D	1101
6	6	0110	14	E	1110
7	7	0111	15	F	1111

介绍了位和字节的相关内容，接下来我们研究 C 用位和字节进行哪些操作。C 有两个操控位的工具。第 1 个工具是一套（6 个）作用于位的按位运算符。第 2 个工具是**字段**（*field*）数据形式，用于访问 int 中的位。下面将简要介绍这些 C 的特性。

15.3　C 按位运算符

C 提供按位逻辑运算符和移位运算符。在下面的例子中，为了方便读者了解位的操作，我们用二进制记数法写出值。但是在实际的程序中不必这样，用一般形式的整型变量或常量即可。例如，在程序中用 25 或 031 或 0x19，而不是 00011001。另外，下面的例子均使用 8 位二进制数，从左往右每位的编号为 7～0。

15.3.1 按位逻辑运算符

4 个按位逻辑运算符都用于整型数据，包括 char。之所以叫作**按位**（*bitwise*）运算，是因为这些操作都是针对每一个位进行，不影响它左右两边的位。不要把这些运算符与常规的逻辑运算符（&&、||和!）混淆，常规的逻辑运算符操作的是整个值。

1. 二进制反码或按位取反：~

一元运算符～把 1 变为 0，把 0 变为 1。如下例子所示：

```
~(10011010)  // 表达式
(01100101)   // 结果值
```

假设 val 的类型是 unsigned char，已被赋值为 2。在二进制中，00000010 表示 2。那么，～val 的值是 11111101，即 253。注意，该运算符不会改变 val 的值，就像 3 * val 不会改变 val 的值一样，val 仍然是 2。但是，该运算符确实创建了一个可以使用或赋值的新值：

```
newval = ~val;
printf("%d", ~val);
```

如果要把 val 的值改为～val，使用下面这条语句：

```
val = ~val;
```

2. 按位与：&

二元运算符&通过逐位比较两个运算对象，生成一个新值。对于每个位，只有两个运算对象中相应的位都为 1 时，结果才为 1（从真/假方面看，只有当两个位都为真时，结果才为真）。因此，对下面的表达式求值：

```
(10010011) & (00111101)   // 表达式
```

由于两个运算对象中编号为 4 和 0 的位都为 1，得：

```
(00010001)   // 结果值
```

C 有一个按位与和赋值结合的运算符：&=。下面两条语句产生的最终结果相同：

```
val &= 0377;
val = val & 0377;
```

3. 按位或：|

二元运算符|，通过逐位比较两个运算对象，生成一个新值。对于每个位，如果两个运算对象中相应的位为 1，结果就为 1（从真/假方面看，如果两个运算对象中相应的一个位为真或两个位都为真，那么结果为真）。因此，对下面的表达式求值：

```
(10010011) | (00111101) // 表达式
```

除了编号为 6 的位，这两个运算对象的其他位至少有一个位为 1，得：

```
(10111111) // 结果值
```

C 有一个按位或和赋值结合的运算符：|=。下面两条语句产生的最终作用相同：

```
val |= 0377;
val = val | 0377;
```

4. 按位异或：^

二元运算符^逐位比较两个运算对象。对于每个位，如果两个运算对象中相应的位一个为 1（但不是两个为 1），结果为 1（从真/假方面看，如果两个运算对象中相应的一个位为真且不是两个为同为 1，那么结

果为真）。因此，对下面表达式求值：

```
(10010011) ^ (00111101) // 表达式
```

编号为 0 的位都是 1，所以结果为 0，得：

```
(10101110)    // 结果值
```

C 有一个按位异或和赋值结合的运算符：^=。下面两条语句产生的最终作用相同：

```
val ^= 0377;
val = val ^ 0377;
```

15.3.2　用法：掩码

按位与运算符常用于**掩码**（*mask*）。所谓掩码指的是一些设置为开（1）或关（0）的位组合。要明白称其为掩码的原因，先来看通过&把一个量与掩码结合后发生什么情况。例如，假设定义符号常量 MASK 为 2（即，二进制形式为 00000010），只有 1 号位是 1，其他位都是 0。下面的语句：

```
flags = flags & MASK;
```

把 flags 中除 1 号位以外的所有位都设置为 0，因为使用按位与运算符（&）任何位与 0 组合都得 0。1 号位的值不变（如果 1 号位是 1，那么 1&1 得 1；如果 1 号位是 0，那么 0&1 也得 0）。这个过程叫作“使用掩码”，因为掩码中的 0 隐藏了 flags 中相应的位。

可以这样类比：把掩码中的 0 看作不透明，1 看作透明。表达式 flags & MASK 相当于用掩码覆盖在 flags 的位组合上，只有 MASK 为 1 的位才可见（见图 15.2）。

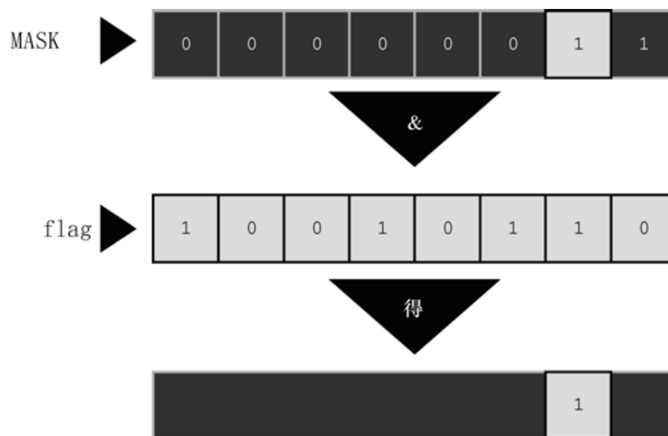


图 15.2　掩码示例

用&=运算符可以简化前面的代码，如下所示：

```
flags &= MASK;
```

下面这条语句是按位与的一种常见用法：

```
ch &= 0xff; /* 或者 ch &= 0377; */
```

前面介绍过 oxff 的二进制形式是 11111111，八进制形式是 0377。这个掩码保持 ch 中最后 8 位不变，其他位都设置为 0。无论 ch 原来是 8 位、16 位或是其他更多位，最终的值都被修改为 1 个 8 位字节。在该例中，掩码的宽度为 8 位。

15.3.3　用法：打开位（设置位）

有时，需要打开一个值中的特定位，同时保持其他位不变。例如，一台 IBM PC 通过向端口发送值来

控制硬件。例如，为了打开内置扬声器，必须打开 1 号位，同时保持其他位不变。这种情况可以使用**按位或**运算符（|）。

以上一节的 flags 和 MASK（只有 1 号位为 1）为例。下面的语句：

```
flags = flags | MASK;
```

把 flags 的 1 号位设置为 1，且其他位不变。因为使用|运算符，任何位与 0 组合，结果都为本身；任何位与 1 组合，结果都为 1。

例如，假设 flags 是 00001111，MASK 是 10110110。下面的表达式：

```
flags | MASK
```

即是：

```
(00001111) | (10110110)    // 表达式
```

其结果为：

```
(10111111)                 // 结果值
```

MASK 中为 1 的位，flags 与其对应的位也为 1。MASK 中为 0 的位，flags 与其对应的位不变。

用|=运算符可以简化上面的代码，如下所示：

```
flags |= MASK;
```

同样，这种方法根据 MASK 中为 1 的位，把 flags 中对应的位设置为 1，其他位不变。

15.3.4　用法：关闭位（清空位）

和打开特定的位类似，有时也需要在不影响其他位的情况下关闭指定的位。假设要关闭变量 flags 中的 1 号位。同样，MASK 只有 1 号位为 1（即，打开）。可以这样做：

```
flags = flags & ~MASK;
```

由于 MASK 除 1 号位为 1 以外，其他位全为 0，所以~MASK 除 1 号位为 0 以外，其他位全为 1。使用&，任何位与 1 组合都得本身，所以这条语句保持 1 号位不变，改变其他各位。另外，使用&，任何位与 0 组合都的 0。所以无论 1 号位的初始值是什么，都将其设置为 0。

例如，假设 flags 是 00001111，MASK 是 10110110。下面的表达式：

```
flags & ~MASK
```

即是：

```
(00001111) & ~(10110110)   // 表达式
```

其结果为：

```
(00001001)                 // 结果值
```

MASK 中为 1 的位在结果中都被设置（清空）为 0。flags 中与 MASK 为 0 的位相应的位在结果中都未改变。

可以使用下面的简化形式：

```
flags &= ~MASK;
```

15.3.5　用法：切换位

切换位指的是打开已关闭的位，或关闭已打开的位。可以使用按位异或运算符（^）切换位。也就是说，假设 b 是一个位（1 或 0），如果 b 为 1，则 1^b 为 0；如果 b 为 0，则 1^b 为 1。另外，无论 b 为 1 还是 0，0^b 均为 b。因此，如果使用^组合一个值和一个掩码，将切换该值与 MASK 为 1 的位相对应的位，该

值与 MASK 为 0 的位相对应的位不变。要切换 flags 中的 1 号位，可以使用下面两种方法：

```
flags = flags ^ MASK;
flags ^= MASK;
```

例如，假设 flags 是 00001111，MASK 是 10110110。表达式：

```
flags ^ MASK
```

即是：

```
(00001111) ^ (10110110)    // 表达式
```

其结果为：

```
(10111001)                 // 结果值
```

flags 中与 MASK 为 1 的位相对应的位都被切换了，MASK 为 0 的位相对应的位不变。

15.3.6　用法：检查位的值

前面介绍了如何改变位的值。有时，需要检查某位的值。例如，flags 中 1 号位是否被设置为 1？不能这样直接比较 flags 和 MASK：

```
if (flags == MASK)
    puts("Wow!"); /* 不能正常工作 */
```

这样做即使 flags 的 1 号位为 1，其他位的值会导致比较结果为假。因此，必须覆盖 flags 中的其他位，只用 1 号位和 MASK 比较：

```
if ((flags & MASK) == MASK)
    puts("Wow!");
```

由于按位运算符的优先级比==低，所以必须在 flags & MASK 周围加上圆括号。

为了避免信息漏过边界，掩码至少要与其覆盖的值宽度相同。

15.3.7　移位运算符

下面介绍 C 的移位运算符。移位运算符向左或向右移动位。同样，我们在示例中仍然使用二进制数，有助于读者理解其工作原理。

1．左移：<<

左移运算符（<<）将其左侧运算对象每一位的值向左移动其右侧运算对象指定的位数。左侧运算对象移出左末端位的值丢失，用 0 填充空出的位置。下面的例子中，每一位都向左移动两个位置：

```
(10001010) << 2    // 表达式
(00101000)         // 结果值
```

该操作产生了一个新的位值，但是不改变其运算对象。例如，假设 stonk 为 1，那么 stonk<<2 为 4，但是 stonk 本身不变，仍为 1。可以使用左移赋值运算符（<<=）来更改变量的值。该运算符将变量中的位向左移动其右侧运算对象给定值的位数。如下例：

```
int stonk = 1;
int onkoo;
onkoo = stonk << 2;    /* 把 4 赋给 onkoo */
stonk <<= 2;           /* 把 stonk 的值改为 4 */
```

2．右移：>>

右移运算符（>>）将其左侧运算对象每一位的值向右移动其右侧运算对象指定的位数。左侧运算对象

移出右末端位的值丢。对于无符号类型，用 0 填充空出的位置；对于有符号类型，其结果取决于机器。空出的位置可用 0 填充，或者用符号位（即，最左端的位）的副本填充：

```
(10001010) >> 2        // 表达式，有符号值
(00100010)             // 在某些系统中的结果值
(10001010) >> 2        // 表达式，有符号值
(11100010)             // 在另一些系统上的结果值
```

下面是无符号值的例子：

```
(10001010) >> 2        // 表达式，无符号值
(00100010)             // 所有系统都得到该结果值
```

每个位向右移动两个位置，空出的位用 0 填充。

右移赋值运算符（>>=）将其左侧的变量向右移动指定数量的位数。如下所示：

```
int sweet = 16;
int ooosw;

ooosw = sweet >> 3;    // ooosw = 2，sweet 的值仍然为 16
sweet >>=3;            // sweet 的值为 2
```

3. 用法：移位运算符

移位运算符针对 2 的幂提供快速有效的乘法和除法：

number << n	number 乘以 2 的 n 次幂
number >> n	如果 number 为非负，则用 number 除以 2 的 n 次幂

这些移位运算符类似于在十进制中移动小数点来乘以或除以 10。

移位运算符还可用于从较大单元中提取一些位。例如，假设用一个 unsigned long 类型的值表示颜色值，低阶位字节储存红色的强度，下一个字节储存绿色的强度，第 3 个字节储存蓝色的强度。随后你希望把每种颜色的强度分别储存在 3 个不同的 unsigned char 类型的变量中。那么，可以使用下面的语句：

```
#define BYTE_MASK 0xff
unsigned long color = 0x002a162f;
unsigned char blue, green, red;
red = color & BYTE_MASK;
green = (color >> 8) & BYTE_MASK;
blue = (color >> 16) & BYTE_MASK;
```

以上代码中，使用右移运算符将 8 位颜色值移动至低阶字节，然后使用掩码技术把低阶字节赋给指定的变量。

15.3.8　编程示例

在第 9 章中，我们用递归的方法编写了一个程序，把数字转换为二进制形式（程序清单 9.8）。现在，要用移位运算符来解决相同的问题。程序清单 15.1 中的程序，读取用户从键盘输入的整数，将该整数和一个字符串地址传递给 itobs()函数（itobs 表示 *interger to binary string*，即整数转换成二进制字符串）。然后，该函数使用移位运算符计算出正确的 1 和 0 的组合，并将其放入字符串中。

程序清单 15.1　binbit.c 程序

```
/* binbit.c -- 使用位操作显示二进制 */
#include <stdio.h>
#include <limits.h>  // 提供 CHAR_BIT 的定义，CHAR_BIT 表示每字节的位数
char * itobs(int, char *);
```

```
void show_bstr(const char *);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];
    int number;

    puts("Enter integers and see them in binary.");
    puts("Non-numeric input terminates program.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("%d is ", number);
        show_bstr(bin_str);
        putchar('\n');
    }
    puts("Bye!");

    return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}

/*4 位一组显示二进制字符串 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i])  /* 不是一个空字符 */
    {
        putchar(str[i]);
        if (++i % 4 == 0 && str[i])
            putchar(' ');
    }
}
```

程序清单 15.1 使用 `limits.h` 中的 `CHAR_BIT` 宏，该宏表示 `char` 中的位数。`sizeof` 运算符返回 `char` 的大小，所以表达式 `CHAE_BIT * sizeof(int)`表示 `int` 类型的位数。`bin_str` 数组的元素个数是 `CHAE_BIT * sizeof(int) + 1`，留出一个位置给末尾的空字符。

`itobs()`函数返回的地址与传入的地址相同，可以把该函数作为 `printf()`的参数。在该函数中，首次执行 `for` 循环时，对 `01 & n` 求值。`01` 是一个八进制形式的掩码，该掩码除 `0` 号位是 `1` 之外，其他所有位都为 `0`。因此，`01 & n` 就是 `n` 最后一位的值。该值为 `0` 或 `1`。但是对数组而言，需要的是**字符**`'0'`或

字符'1'。该值加上'0'即可完成这种转换（假设按顺序编码的数字，如 ASCII）。其结果存放在数组中倒数第 2 个元素中（最后一个元素用来存放空字符）。

顺带一提，用 1 & n 或 01 & n 都可以。我们用八进制 1 而不是十进制 1，只是为了更接近计算机的表达方式。

然后，循环执行 i--和 n >>= 1。i--移动到数组的前一个元素，n >>= 1 使 n 中的所有位向右移动一个位置。进入下一轮迭代时，循环中处理的是 n 中新的最右端的值。然后，把该值储存在倒数第 3 个元素中，以此类推。itobs()函数用这种方式从右往左填充数组。

可以使用 printf()或 puts()函数显示最终的字符串，但是程序清单 15.1 中定义了 show_bstr()函数，以 4 位一组打印字符串，方便阅读。

下面的该程序的运行示例：

```
Enter integers and see them in binary.
Non-numeric input terminates program.
7
7 is 0000 0000 0000 0000 0000 0000 0000 0111
2013
2013 is 0000 0000 0000 0000 0000 0111 1101 1101
-1
-1 is 1111 1111 1111 1111 1111 1111 1111 1111
32123
32123 is 0000 0000 0000 0000 0111 1101 0111 1011
q
Bye!
```

15.3.9　另一个例子

我们来看另一个例子。这次要编写的函数用于切换一个值中的后 n 位，待处理值和 n 都是函数的参数。

～运算符切换一个字节的所有位，而不是选定的少数位。但是，^运算符（按位异或）可用于切换单个位。假设创建了一个掩码，把后 n 位设置为 1，其余位设置为 0。然后使用^组合掩码和待切换的值便可切换该值的最后 n 位，而且其他位不变。方法如下：

```
int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits–– > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }
    return num ^ mask;
}
```

while 循环用于创建所需的掩码。最初，mask 的所有位都为 0。第 1 轮循环将 mask 的 0 号位设置为 1。然后第 2 轮循环将 mask 的 1 号位设置为 1，以此类推。循环 bits 次，mask 的后 bits 位就都被设置为 1。最后，num ^ mask 运算即得所需的结果。

我们把这个函数放入前面的程序中，测试该函数。如程序清单 15.2 所示。

程序清单 15.2　invert4.c 程序

```
/* invert4.c -- 使用位操作显示二进制 */
#include <stdio.h>
#include <limits.h>
char * itobs(int, char *);
void show_bstr(const char *);
int invert_end(int num, int bits);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(int) + 1];

    int number;

    puts("Enter integers and see them in binary.");
    puts("Non-numeric input terminates program.");
    while (scanf("%d", &number) == 1)
    {
        itobs(number, bin_str);
        printf("%d is\n", number);
        show_bstr(bin_str);
        putchar('\n');
        number = invert_end(number, 4);
        printf("Inverting the last 4 bits gives\n");
        show_bstr(itobs(number, bin_str));
        putchar('\n');
    }
    puts("Bye!");

    return 0;
}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}

/* 以 4 位为一组，显示二进制字符串 */
void show_bstr(const char * str)
{
    int i = 0;

    while (str[i])  /* 不是空字符 */
    {
        putchar(str[i]);
        if (++i % 4 == 0 && str[i])
            putchar(' ');
```

```
    }
}

int invert_end(int num, int bits)
{
    int mask = 0;
    int bitval = 1;

    while (bits-- > 0)
    {
        mask |= bitval;
        bitval <<= 1;
    }

    return num ^ mask;
}
```

下面是该程序的一个运行示例：

```
Enter integers and see them in binary.
Non-numeric input terminates program.
7
7 is
0000 0000 0000 0000 0000 0000 0000 0111
Inverting the last 4 bits gives
0000 0000 0000 0000 0000 0000 0000 1000
12541
12541 is
0000 0000 0000 0000 0011 0000 1111 1101
Inverting the last 4 bits gives
0000 0000 0000 0000 0011 0000 1111 0010
q
Bye!
```

15.4　位字段

操控位的第 2 种方法是**位字段**（*bit field*）。位字段是一个 `signed int` 或 `unsigned int` 类型变量中的一组相邻的位（C99 和 C11 新增了`_Bool` 类型的位字段）。位字段通过一个结构声明来建立，该结构声明为每个字段提供标签，并确定该字段的宽度。例如，下面的声明建立了一个 4 个 1 位的字段：

```
struct {
    unsigned int autfd : 1;
    unsigned int bldfc : 1;
    unsigned int undln : 1;
    unsigned int itals : 1;
} prnt;
```

根据该声明，`prnt` 包含 4 个 1 位的字段。现在，可以通过普通的结构成员运算符（`.`）单独给这些字段赋值：

```
prnt.itals = 0;
prnt.undln = 1;
```

由于每个字段恰好为 1 位，所以只能为其赋值 `1` 或 `0`。变量 `prnt` 被储存在 `int` 大小的内存单元中，但是在本例中只使用了其中的 4 位。

带有位字段的结构提供一种记录设置的方便途径。许多设置（如，字体的粗体或斜体）就是简单的二

选一。例如，开或关、真或假。如果只需要使用 1 位，就不需要使用整个变量。内含位字段的结构允许在一个存储单元中储存多个设置。

有时，某些设置也有多个选择，因此需要多位来表示。这没问题，字段不限制 1 位大小。可以使用如下的代码：

```
struct {
    unsigned int code1 : 2;
    unsigned int code2 : 2;
    unsigned int code3 : 8;
} prcode;
```

以上代码创建了两个 2 位的字段和一个 8 位的字段。可以这样赋值：

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```

但是，要确保所赋的值不超出字段可容纳的范围。

如果声明的总位数超过了一个 `unsigned int` 类型的大小会怎样？会用到下一个 `unsigned int` 类型的存储位置。一个字段不允许跨越两个 `unsigned int` 之间的边界。编译器会自动移动跨界的字段，保持 `unsigned int` 的边界对齐。一旦发生这种情况，第 1 个 `unsigned int` 中会留下一个未命名的“洞”。

可以用未命名的字段宽度“填充”未命名的“洞”。使用一个宽度为 `0` 的未命名字段迫使下一个字段与下一个整数对齐：

```
struct {
    unsigned int field1    : 1 ;
    unsigned int           : 2 ;
    unsigned int field2    : 1 ;
    unsigned int           : 0 ;
    unsigned int field3    : 1 ;
} stuff;
```

这里，在 `stuff.field1` 和 `stuff.field2` 之间，有一个 2 位的空隙；`stuff.field3` 将储存在下一个 `unsigned int` 中。

字段储存在一个 `int` 中的顺序取决于机器。在有些机器上，存储的顺序是从左往右，而在另一些机器上，是从右往左。另外，不同的机器中两个字段边界的位置也有区别。由于这些原因，位字段通常都不容易移植。尽管如此，有些情况却要用到这种不可移植的特性。例如，以特定硬件设备所用的形式储存数据。

15.4.1　位字段示例

通常，把位字段作为一种更紧凑储存数据的方式。例如，假设要在屏幕上表示一个方框的属性。为简化问题，我们假设方框具有如下属性：

- 方框是透明的或不透明的；
- 方框的填充色选自以下调色板：黑色、红色、绿色、黄色、蓝色、紫色、青色或白色；
- 边框可见或隐藏；
- 边框颜色与填充色使用相同的调色板；
- 边框可以使用实线、点线或虚线样式。

可以使用单独的变量或全长（*full-sized*）结构成员来表示每个属性，但是这样做有些浪费位。例如，只需 1 位即可表示方框是透明还是不透明；只需 1 位即可表示边框是显示还是隐藏。8 种颜色可以用 3 位单元的 8 个可能的

值来表示，而 3 种边框样式也只需 2 位单元即可表示。总共 10 位就足够表示方框的 5 个属性设置。

一种方案是：一个字节储存方框内部（透明和填充色）的属性，一个字节储存方框边框的属性，每个字节中的空隙用未命名字段填充。struct box_props 声明如下：

```
struct box_props {
    bool opaque                 : 1 ;
    unsigned int fill_color     : 3 ;
    unsigned int                : 4 ;
    bool show_border            : 1 ;
    unsigned int border_color   : 3 ;
    unsigned int border_style   : 2 ;
    unsigned int                : 2 ;
};
```

加上未命名的字段，该结构共占用 16 位。如果不使用填充，该结构占用 10 位。但是要记住，C 以 unsigned int 作为位字段结构的基本布局单元。因此，即使一个结构唯一的成员是 1 位字段，该结构的大小也是一个 unsigned int 类型的大小，unsigned int 在我们的系统中是 32 位。另外，以上代码假设 C99 新增的_Bool 类型可用，在 stdbool.h 中，bool 是_Bool 的别名。

对于 opaque 成员，1 表示方框不透明，0 表示透明。show_border 成员也用类似的方法。对于颜色，可以用简单的 RGB（即 red-green-blue 的缩写）表示。这些颜色都是三原色的混合。显示器通过混合红、绿、蓝像素来产生不同的颜色。在早期的计算机色彩中，每个像素都可以打开或关闭，所以可以使用用 1 位来表示三原色中每个二进制颜色的亮度。常用的顺序是，左侧位表示蓝色亮度、中间位表示绿色亮度、右侧位表示红色亮度。表 15.3 列出了这 8 种可能的组合。fill_color 成员和 border_color 成员可以使用这些组合。最后，border_style 成员可以使用 0、1、2 来表示实线、点线和虚线样式。

表 15.3 简单的颜色表示

位组合	十进制	颜色
000	0	黑色
001	1	红色
010	2	绿色
011	3	黄色
100	4	蓝色
101	5	紫色
110	6	青色
111	7	白色

程序清单 15.3 中的程序使用 box_props 结构，该程序用#define 创建供结构成员使用的符号常量。注意，只打开一位即可表示三原色之一。其他颜色用三原色的组合来表示。例如，紫色由打开的蓝色位和红色位组成，所以，紫色可表示为 BLUE|RED。

程序清单 15.3 fields.c 程序

```
/* fields.c -- 定义并使用字段 */
#include <stdio.h>
#include <stdbool.h>   // C99 定义了 bool、true、false

/* 线的样式 */
#define SOLID   0
#define DOTTED  1
#define DASHED  2
```

```
/* 三原色 */
#define BLUE    4
#define GREEN   2
#define RED     1
/* 混合色 */
#define BLACK   0
#define YELLOW  (RED | GREEN)
#define MAGENTA (RED | BLUE)
#define CYAN    (GREEN | BLUE)
#define WHITE   (RED | GREEN | BLUE)

const char * colors[8] = { "black", "red", "green", "yellow",
"blue", "magenta", "cyan", "white" };

struct box_props {
    bool opaque : 1;       // 或者 unsigned int （C99 以前）
    unsigned int fill_color : 3;
    unsigned int : 4;
    bool show_border : 1;  // 或者 unsigned int （C99 以前）
    unsigned int border_color : 3;
    unsigned int border_style : 2;
    unsigned int : 2;
};

void show_settings(const struct box_props * pb);

int main(void)
{
    /* 创建并初始化 box_props 结构 */
    struct box_props box = { true, YELLOW, true, GREEN, DASHED };

    printf("Original box settings:\n");
    show_settings(&box);

    box.opaque = false;
    box.fill_color = WHITE;
    box.border_color = MAGENTA;
    box.border_style = SOLID;
    printf("\nModified box settings:\n");
    show_settings(&box);

    return 0;
}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
        pb->opaque == true ? "opaque" : "transparent");
    printf("The fill color is %s.\n", colors[pb->fill_color]);
    printf("Border %s.\n",
        pb->show_border == true ? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf("The border style is ");
    switch (pb->border_style)
    {
```

```
        case SOLID:  printf("solid.\n"); break;
        case DOTTED: printf("dotted.\n"); break;
        case DASHED: printf("dashed.\n"); break;
        default:     printf("unknown type.\n");
        }
    }
```

下面是该程序的输出：

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Modified box settings:
Box is transparent.
The fill color is white.
Border shown.
The border color is magenta.
The border style is solid.
```

该程序要注意几个要点。首先，初始化位字段结构与初始化普通结构的语法相同：

```
struct box_props box = {YES, YELLOW , YES, GREEN, DASHED};
```

类似地，也可以给位字段成员赋值：

```
box.fill_color = WHITE;
```

另外，switch 语句中也可以使用位字段成员，甚至还可以把位字段成员用作数组的下标：

```
printf("The fill color is %s.\n", colors[pb->fill_color]);
```

注意，根据 colors 数组的定义，每个索引对应一个表示颜色的字符串，而每种颜色都把索引值作为该颜色的数值。例如，索引 1 对应字符串"red"，枚举常量 red 的值是 1。

15.4.2 位字段和按位运算符

在同类型的编程问题中，位字段和按位运算符是两种可替换的方法，用哪种方法都可以。例如，前面的例子中，使用和 unsigned int 类型大小相同的结构储存图形框的信息。也可使用 unsigned int 变量储存相同的信息。如果不想用结构成员表示法来访问不同的部分，也可以使用按位运算符来操作。一般而言，这种方法比较麻烦。接下来，我们来研究这两种方法（程序中使用了这两种方法，仅为了解释它们的区别，我们并不鼓励这样做）。

可以通过一个联合把结构方法和位方法放在一起。假定声明了 struct box_props 类型，然后这样声明联合：

```
union Views /* 把数据看作结构或unsigned short 类型的变量 */
{
    struct box_props st_view;
    unsigned short us_view;
};
```

在某些系统中，unsigned int 和 box_props 类型的结构都占用 16 位内存。但是，在其他系统中（例如我们使用的系统），unsigned int 和 box_props 都是 32 位。无论哪种情况，通过联合，都可以使用 st_view 成员把一块内存看作是一个结构，或者使用 us_view 成员把相同的内存块看作是一个 unsigned short。结构的哪一个位字段与 unsigned short 中的哪一位对应？这取决于实现和硬件。下面的程序示例假设从字节的低阶位端到高阶位端载入结构。也就是说，结构中的第 1 个位字段对应计算机字的 0 号位（为简化起见，图 15.3 以 16 位单元演示了这种情况）。

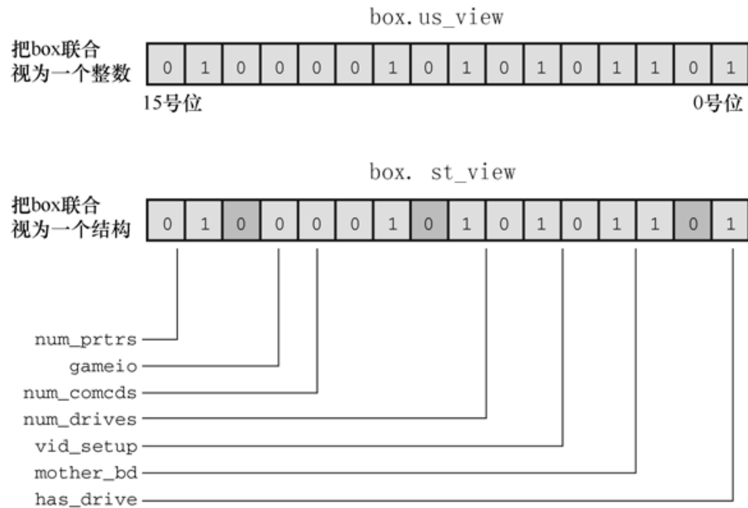


图 15.3　作为整数和结构的联合

程序清单 15.4 使用 `Views` 联合来比较位字段和按位运算符这两种方法。在该程序中，`box` 是 `View` 联合，所以 `box.st_view` 是一个使用位字段的 `box_props` 类型的结构，`box.us_view` 把相同的数据看作是一个 `unsigned short` 类型的变量。联合只允许初始化第 1 个成员，所以初始化值必须与结构相匹配。该程序分别通过两个函数显示 `box` 的属性，一个函数接受一个结构，一个函数接受一个 `unsigned short` 类型的值。这两种方法都能访问数据，但是所用的技术不同。该程序还使用了本章前面定义的 `itobs()`函数，以二进制字符串形式显示数据，以便读者查看每个位的开闭情况。

程序清单 15.4　dualview.c 程序

```
/* dualview.c -- 位字段和按位运算符 */
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>
/* 位字段符号常量 */
/* 边框线样式     */
#define SOLID      0
#define DOTTED     1
#define DASHED     2
/* 三原色 */
#define BLUE       4
#define GREEN      2
#define RED        1
/* 混合颜色 */
#define BLACK      0
#define YELLOW     (RED | GREEN)
#define MAGENTA    (RED | BLUE)
#define CYAN       (GREEN | BLUE)
#define WHITE      (RED | GREEN | BLUE)

/* 按位方法中用到的符号常量 */
#define OPAQUE          0x1
#define FILL_BLUE       0x8
#define FILL_GREEN      0x4
#define FILL_RED        0x2
#define FILL_MASK       0xE
#define BORDER          0x100
#define BORDER_BLUE     0x800
```

```
#define BORDER_GREEN   0x400
#define BORDER_RED0x   200
#define BORDER_MASK    0xE00
#define B_SOLID        0
#define B_DOTTED       0x1000
#define B_DASHED       0x2000
#define STYLE_MASK0x   3000

const char * colors[8] = { "black", "red", "green", "yellow", "blue", "magenta",
"cyan", "white" };

struct box_props {

    bool opaque                 : 1;
    unsigned int fill_color     : 3;
    unsigned int                : 4;
    bool show_border            : 1;
    unsigned int border_color   : 3;
    unsigned int border_style   : 2;
    unsigned int                : 2;
};

union Views   /* 把数据看作结构或 unsigned short 类型的变量 */
{
    struct box_props st_view;
    unsigned short us_view;
};

void show_settings(const struct box_props * pb);
void show_settings1(unsigned short);
char * itobs(int n, char * ps);

int main(void)
{
    /* 创建 Views 联合，并初始化 initialize struct box view */
    union Views box = { { true, YELLOW, true, GREEN, DASHED } };
    char bin_str[8 * sizeof(unsigned int) + 1];

    printf("Original box settings:\n");
    show_settings(&box.st_view);
    printf("\nBox settings using unsigned int view:\n");
    show_settings1(box.us_view);

    printf("bits are %s\n",
        itobs(box.us_view, bin_str));
    box.us_view &= ~FILL_MASK;                  /* 把表示填充色的位清 0 */
    box.us_view |= (FILL_BLUE | FILL_GREEN);    /* 重置填充色 */
    box.us_view ^= OPAQUE;                      /* 切换是否透明的位 */
    box.us_view |= BORDER_RED;                  /* 错误的方法 */
    box.us_view &= ~STYLE_MASK;                 /* 把样式的位清 0 */
    box.us_view |= B_DOTTED;                    /* 把样式设置为点 */
    printf("\nModified box settings:\n");
    show_settings(&box.st_view);
    printf("\nBox settings using unsigned int view:\n");
    show_settings1(box.us_view);
    printf("bits are %s\n",
        itobs(box.us_view, bin_str));

    return 0;
```

```
}

void show_settings(const struct box_props * pb)
{
    printf("Box is %s.\n",
        pb->opaque == true ? "opaque" : "transparent");
    printf("The fill color is %s.\n", colors[pb->fill_color]);
    printf("Border %s.\n",
        pb->show_border == true ? "shown" : "not shown");
    printf("The border color is %s.\n", colors[pb->border_color]);
    printf("The border style is ");
    switch (pb->border_style)
    {
    case SOLID:  printf("solid.\n"); break;
    case DOTTED: printf("dotted.\n"); break;
    case DASHED: printf("dashed.\n"); break;
    default:     printf("unknown type.\n");
    }
}

void show_settings1(unsigned short us)
{
    printf("box is %s.\n",
        (us & OPAQUE) == OPAQUE ? "opaque" : "transparent");
    printf("The fill color is %s.\n",
        colors[(us >> 1) & 07]);
    printf("Border %s.\n",
        (us & BORDER) == BORDER ? "shown" : "not shown");
    printf("The border style is ");
    switch (us & STYLE_MASK)
    {
    case B_SOLID  : printf("solid.\n"); break;
    case B_DOTTED : printf("dotted.\n"); break;
    case B_DASHED : printf("dashed.\n"); break;
    default       : printf("unknown type.\n");
    }
    printf("The border color is %s.\n",
        colors[(us >> 9) & 07]);

}

char * itobs(int n, char * ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(int);

    for (i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';

    return ps;
}
```

下面是该程序的输出：

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.
Box settings using unsigned int view:
```

```
box is opaque.
The fill color is yellow.
Border shown.
The border style is dashed.
The border color is green.
bits are 000000000000000000010010100000111
Modified box settings:
Box is transparent.
The fill color is cyan.
Border shown.
The border color is yellow.
The border style is dotted.
Box settings using unsigned int view:
box is transparent.
The fill color is cyan.
Border not shown.
The border style is dotted.
The border color is yellow.
bits are 00000000000000000001011100001100
```

这里要讨论几个要点。位字段视图和按位视图的区别是，按位视图需要位置信息。例如，程序中使用 BLUE 表示蓝色，该符号常量的数值为 4。但是，由于结构排列数据的方式，实际储存蓝色设置的是 3 号位（位的编号从 0 开始，参见图 15.1），而且储存边框为蓝色的设置是 11 号位。因此，该程序定义了一些新的符号常量：

```
#define FILL_BLUE      0x8
#define BORDER_BLUE    0x800
```

这里，0x8 是 3 号位为 1 时的值，0x800 是 11 号位为 1 时的值。可以使用第 1 个符号常量设置填充色的蓝色位，用第 2 个符号常量设置边框颜色的蓝色位。用十六进制记数法更容易看出要设置二进制的哪一位，由于十六进制的每一位代表二进制的 4 位，那么 0x8 的位组合是 1000，而 0x800 的位组合是 100000000000，0x800 的位组合比 0x8 后面多 8 个 0。但是以等价的十进制来看就没那么明显，0x8 是 8，0x800 是 2048。

如果值是 2 的幂，那么可以使用左移运算符来表示值。例如，可以用下面的#define 分别替换上面的#define：

```
#define FILL_BLUE      1<<3
#define BORDER_BLUE    1<<11
```

这里，<<的右侧是 2 的指数，也就是说，0x8 是 2^3，0x800 是 2^{11}。同样，表达式 1<<n 指的是第 n 位为 1 的整数。1<<11 是常量表达式，在编译时求值。

可以使用枚举代替#defined 创建符号常量。例如，可以这样做：

```
enum { OPAQUE = 0x1, FILL_BLUE = 0x8, FILL_GREEN = 0x4, FILL_RED = 0x2,
       FILL_MASK = 0xE, BORDER = 0x100, BORDER_BLUE = 0x800,
       BORDER_GREEN = 0x400, BORDER_RED = 0x200, BORDER_MASK = 0xE00,
       B_DOTTED = 0x1000, B_DASHED = 0x2000, STYLE_MASK = 0x3000};
```

如果不想创建枚举变量，就不用在声明中使用标记。

注意，按位运算符改变设置更加复杂。例如，要设置填充色为青色。只打开蓝色位和绿色位是不够的：

```
box.us_view |= (FILL_BLUE | FILL_GREEN); /* 重置填充色 */
```

问题是该颜色还依赖于红色位的设置。如果已经设置了该位（比如对于黄色），这行代码保留了红色位的设置，而且还设置了蓝色位和绿色位，结果是产生白色。解决这个问题最简单的方法是在设置新值前关闭所有的颜色位。因此，程序中使用了下面两行代码：

```
box.us_view &= ~FILL_MASK;                 /* 把表示填充色的位清 0 */
box.us_view |= (FILL_BLUE | FILL_GREEN);   /* 重置填充色 */
```

如果不先关闭所有的相关位，程序中演示了这种情况：

```
box.us_view |= BORDER_RED;      /* 错误的方法 */
```

因为BORDER_GREEN位已经设置过了，所以结果颜色是BORDER_GREEN | BORDER_RED，被解释为黄色。

这种情况下，位字段版本更简单：

```
box.st_view.fill_color = CYAN; /*等价的位字段方法 */
```

这种方法不用先清空所有的位。而且，使用位字段成员时，可以为边框和框内填充色使用相同的颜色值。但是用按位运算符的方法则要使用不同的值（这些值反映实际位的位置）。

其次，比较下面两个打印语句：

```
printf("The border color is %s.\n", colors[pb->border_color]);
printf("The border color is %s.\n", colors[(us >> 9) & 07]);
```

第 1 条语句中，表达式 pb->border_color 的值在 0～7 的范围内，所以该表达式可用作 colors 数组的索引。用按位运算符获得相同的信息更加复杂。一种方法是使用 ui>>9 把边框颜色右移至最右端（0 号位～2 号位），然后把该值与掩码 07 组合，关闭除了最右端 3 位以外所有的位。这样结果也在 0～7 的范围内，可作为 colors 数组的索引。

警告

位字段和位的位置之间的相互对应因实现而异。例如，在早期的 Macintosh PowerPC 上运行程序清单 15.4，输出如下:

```
Original box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is transparent.
The fill color is black.
Border not shown.
The border style is solid.
The border color is black.
bits are 10110000101010000000000000000000

Modified box settings:
Box is opaque.
The fill color is yellow.
Border shown.
The border color is green.
The border style is dashed.

Box settings using unsigned int view:
box is opaque.
The fill color is cyan.
Border shown.
The border style is dotted.
The border color is red.
bits are 10110000101010000001001000001101
```

该输出的二进制位与程序示例 15.4 不同，Macintosh PowerPC 把结构载入内存的方式不同。特别是，它把第 1 位字段载入最高阶位，而不是最低阶位。所以结构表示法储存在前 16 位（与 PC 中的顺序不同），而 unsigned int 表示法则储存在后 16 位。因此，对于 Macintosh，程序清单 15.4 中关于位的位置的假设是错误的，使用按位运算符改变透明设置和填充色设置时，也弄错了位。

15.5　对齐特性（C11）

C11 的对齐特性比用位填充字节更自然，它们还代表了 C 在处理硬件相关问题上的能力。在这种上下文中，对齐指的是如何安排对象在内存中的位置。例如，为了效率最大化，系统可能要把一个 double 类型的值储存在 4 字节内存地址上，但却允许把 char 储存在任意地址。大部分程序员都对对齐不以为然。但是，有些情况又受益于对齐控制。例如，把数据从一个硬件位置转移到另一个位置，或者调用指令同时操作多个数据项。

_Alignof 运算符给出一个类型的对齐要求，在关键字_Alignof 后面的圆括号中写上类型名即可：

```
size_t d_align = _Alignof(float);
```

假设 d_align 的值是 4，意思是 float 类型对象的对齐要求是 4。也就是说，4 是储存该类型值相邻地址的字节数。一般而言，对齐值都应该是 2 的非负整数次幂。较大的对齐值被称为 stricter 或 stronger，较小的对齐值被称为 weaker。

可以使用_Alignas 说明符指定一个变量或类型的对齐值。但是，不应该要求该值小于基本对齐值。例如，如果 float 类型的对齐要求是 4，不要请求其对齐值是 1 或 2。该说明符用作声明的一部分，说明符后面的圆括号内包含对齐值或类型：

```
_Alignas(double) char c1;
_Alignas(8) char c2;
unsigned char _Alignas(long double) c_arr[sizeof(long double)];
```

注意

撰写本书时，Clang（3.2 版本）要求_Alignas(type)说明符在类型说明符后面，如上面第 3 行代码所示。但是，无论_Alignas(type)说明符在类型说明符的前面还是后面，GCC 4.7.3 都能识别，后来 Clang 3.3 版本也支持了这两种顺序。

程序清单 15.5 中的程序演示了_Alignas 和_Alignof 的用法。

程序清单 15.5　align.c 程序

```
//  align.c -- 使用 _Alignof 和 _Alignas （C11）

#include <stdio.h>
int main(void)
{
    double dx;
    char ca;
    char cx;
    double dz;
    char cb;
    char _Alignas(double)  cz;

    printf("char alignment:   %zd\n", _Alignof(char));
    printf("double alignment: %zd\n", _Alignof(double));
    printf("&dx: %p\n", &dx);
    printf("&ca: %p\n", &ca);
    printf("&cx: %p\n", &cx);
    printf("&dz: %p\n", &dz);
```

```
    printf("&cb: %p\n", &cb);
    printf("&cz: %p\n", &cz);

    return 0;
}
```

该程序的输出如下：

```
char alignment:   1
double alignment: 8
&dx: 0x7fff5fbff660
&ca: 0x7fff5fbff65f
&cx: 0x7fff5fbff65e
&dz: 0x7fff5fbff650
&cb: 0x7fff5fbff64f
&cz: 0x7fff5fbff648
```

在我们的系统中，double 的对齐值是 8，这意味着地址的类型对齐可以被 8 整除。以 0 或 8 结尾的十六进制地址可被8整除。这就是地址常用两个double类型的变量和char类型的变量cz（该变量是double对齐值）。因为 char 的对齐值是 1，对于普通的 char 类型变量，编译器可以使用任何地址。

在程序中包含 stdalign.h 头文件后，就可以把 alignas 和 alignof 分别作为_Alignas 和_Alignof 的别名。这样做可以与 C++关键字匹配。

C11 在 stdlib.h 库还添加了一个新的内存分配函数，用于对齐动态分配的内存。该函数的原型如下：

```
void *aligned_alloc(size_t alignment, size_t size);
```

第 1 个参数代表指定的对齐，第 2 个参数是所需的字节数，其值应是第 1 个参数的倍数。与其他内存分配函数一样，要使用 free()函数释放之前分配的内存。

15.6　关键概念

C 区别于许多高级语言的特性之一是访问整数中单独位的能力。该特性通常是与硬件设备和操作系统交互的关键。

C 有两种访问位的方法。一种方法是通过按位运算符，另一种方法是在结构中创建位字段。

C11 新增了检查内存对齐要求的功能，而且可以指定比基本对齐值更大的对齐值。

通常（但不总是），使用这些特性的程序仅限于特定的硬件平台或操作系统，而且设计为不可移植的。

15.7　本章小结

计算硬件与二进制记数系统密不可分，因为二进制数的 1 和 0 可用于表示计算机内存和寄存器中位的开闭状态。虽然 C 不允许以二进制形式书写数字，但是它识别与二进制相关的八进制和十六进制记数法。正如每个二进制数字表示 1 位一样，每个八进制位代表 3 位，每个十六进制位代表 4 位。这种关系使得二进制转为八进制或十六进制较为简单。

C 提供多种按位运算符，之所以称为按位是因为它们单独操作一个值中的每个位。～运算符将其运算对象的每一位取反，将 1 转为 0，0 转为 1。按位与运算符（&）通过两个运算对象形成一个值。如果两运算对象中相同号位都为 1，那么该值中对应的位为 1；否则，该位为 0。按位或运算符（|）同样通过两个运算对象形成一个值。如果两运算对象中相同号位有一个为 1 或都为 1，那么该值中对应的位为 1；否则，该位为 0。按位异或运算符（^）也有类似的操作，只有两运算对象中相同号位有一个为 1 时，结果值中对

应的位才为1。

C还有左移（<<）和右移（>>）运算符。这两个运算符使位组合中的所有位都向左或向右移动指定数量的位，以形成一个新值。对于左移运算符，空出的位置设为 0。对于右移运算符，如果是无符号类型的值，空出的位设为0；如果是有符号类型的值，右移运算符的行为取决于实现。

可以在结构中使用位字段操控一个值中的单独位或多组位。具体细节因实现而异。

可以使用_Alignas 强制执行数据存储区上的对齐要求。

这些位工具帮助C程序处理硬件问题，因此它们通常用于依赖实现的场合中。

15.8 复习题

复习题的参考答案在附录 A 中。

1. 把下面的十进制转换为二进制：

   a. 3

   b. 13

   c. 59

   d. 119

2. 将下面的二进制值转换为十进制、八进制和十六进制的形式：

   a. 00010101

   b. 01010101

   c. 01001100

   d. 10011101

3. 对下面的表达式求值，假设每个值都为8位：

   a. ~3

   b. 3 & 6

   c. 3 | 6

   d. 1 | 6

   e. 3 ^ 6

   f. 7 >> 1

   g. 7 << 2

4. 对下面的表达式求值，假设每个值都为8位：

   a. ~0

   b. !0

   c. 2 & 4

   d. 2 && 4

   e. 2 | 4

   f. 2 || 4

g. 5 << 3

5. 因为 ASCII 码只使用最后 7 位，所以有时需要用掩码关闭其他位，其相应的二进制掩码是什么？分别用十进制、八进制和十六进制来表示这个掩码。

6. 程序清单 15.2 中，可以把下面的代码：

```
while (bits-- > 0)
{
    mask |= bitval;
    bitval <<= 1;
}
```

替换成：

```
while (bits-- > 0)
{
    mask += bitval;
    bitval *= 2;
}
```

程序照常工作。这是否意味着*=2 等同于<<=1？+=是否等同于|=？

7. a．Tinkerbell 计算机有一个硬件字节可读入程序。该字节包含以下信息：

位	含义
0～1	1.4MB 软盘驱动器的数量
2	未使用
3～4	CD-ROM 驱动器数量
5	未使用
6～7	硬盘驱动器数量

Tinkerbell 和 IBM PC 一样，从右往左填充结构位字段。创建一个适合存放这些信息的位字段模板。

b．Klinkerbell 与 Tinkerbell 类似，但是它从左往右填充结构位字段。请为 Klinkerbell 创建一个相应的位字段模板。

15.9　编程练习

1. 编写一个函数，把二进制字符串转换为一个数值。例如，有下面的语句：

```
char * pbin = "01001001";
```

那么把 pbin 作为参数传递给该函数后，它应该返回一个 int 类型的值 25。

2. 编写一个程序，通过命令行参数读取两个二进制字符串，对这两个二进制数使用～运算符、&运算符、|运算符和^运算符，并以二进制字符串形式打印结果（如果无法使用命令行环境，可以通过交互式让程序读取字符串）。

3. 编写一个函数，接受一个 int 类型的参数，并返回该参数中打开位的数量。在一个程序中测试该函数。

4. 编写一个程序，接受两个 int 类型的参数：一个是值；一个是位的位置。如果指定位的位置为 1，该函数返回 1；否则返回 0。在一个程序中测试该函数。

5. 编写一个函数，把一个 unsigned int 类型值中的所有位向左旋转指定数量的位。例如，

rotate_l(x, 4)把 x 中所有位向左移动 4 个位置，而且从最左端移出的位会重新出现在右端。也就是说，把高阶位移出的位放入低阶位。在一个程序中测试该函数。

6. 设计一个位字段结构以储存下面的信息。

   字体 ID：0～255 之间的一个数；

   字体大小：0～127 之间的一个数；

   对齐：0～2 之间的一个数，表示左对齐、居中、右对齐；

   加粗：开（1）或闭（0）；

   斜体：开（1）或闭（0）；

   在一个程序中使用该结构来打印字体参数，并使用循环菜单来让用户改变参数。例如，该程序的一个运行示例如下：

```
  ID SIZE ALIGNMENT   B   I   U
   1   12   left     off off off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
s
Enter font size (0-127): 36

  ID SIZE ALIGNMENT   B   I   U
   1   36   left     off off off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
a
Select alignment:
l)left   c)center   r)right
r

  ID SIZE ALIGNMENT   B   I   U
   1   36  right     off off off
f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
i

  ID SIZE ALIGNMENT   B   I   U
   1   36  right     off  on off

f)change font    s)change size    a)change alignment
b)toggle bold    i)toggle italic  u)toggle underline
q)quit
q
Bye!
```

   该程序要使用按位与运算符（&）和合适的掩码来把字体 ID 和字体大小信息转换到指定的范围内。

7. 编写一个与编程练习 6 功能相同的程序，使用 unsigned long 类型的变量储存字体信息，并且使用按位运算符而不是位成员来管理这些信息。

第 16 章

C 预处理器和 C 库

本章介绍以下内容：

- 预处理指令：#define、#include、#ifdef、#else、#endif、#ifndef、#if、#elif、#line、#error、#pragma
- 关键字：_Generic、_Noreturn、_Static_assert
- 函数/宏：sqrt()、atan()、atan2()、exit()、atexit()、assert()、memcpy()、memmove()、va_start()、va_arg()、va_copy()、va_end()
- C 预处理器的其他功能
- 通用选择表达式
- 内联函数
- C 库概述和一些特殊用途的方便函数

C 语言建立在适当的关键字、表达式、语句以及使用它们的规则上。然而，C 标准不仅描述 C 语言，还描述如何执行 C 预处理器、C 标准库有哪些函数，以及详述这些函数的工作原理。本章将介绍 C 预处理器和 C 库，我们先从 C 预处理器开始。

C 预处理器在程序执行之前查看程序（故称之为**预处理器**）。根据程序中的预处理器指令，预处理器把符号缩写替换成其表示的内容。预处理器可以包含程序所需的其他文件，可以选择让编译器查看哪些代码。预处理器并不知道 C。基本上它的工作是把一些文本转换成另外一些文本。这样描述预处理器无法体现它的真正效用和价值，我们将在本章举例说明。前面的程序示例中也有很多#define 和#include 的例子。下面，我们先总结一下已学过的预处理指令，再介绍一些新的知识点。

16.1 翻译程序的第一步

在预处理之前，编译器必须对该程序进行一些翻译处理。首先，编译器把源代码中出现的字符映射到源字符集。该过程处理多字节字符和**三字符序列**——字符扩展让 C 更加国际化（详见附录 B“参考资料 VII，扩展字符支持”）。

第二，编译器定位每个反斜杠后面跟着换行符的实例，并删除它们。也就是说，把下面两个物理行（*physical line*）：

```
printf("That's wond\
        erful!\n");
```

转换成一个**逻辑行**（*logical line*）：

```
printf("That's wonderful\n!");
```

注意，在这种场合中，“换行符”的意思是通过按下 Enter 键在源代码文件中换行所生成的字符，而不是指符号表征\n。

由于预处理表达式的长度必须是一个逻辑行，所以这一步为预处理器做好了准备工作。一个逻辑行可以是多个物理行。

第三，编译器把文本划分成预处理记号序列、空白序列和注释序列（记号是由空格、制表符或换行符分隔的项，详见 16.2.1）。这里要注意的是，编译器将用一个空格字符替换每一条注释。因此，下面的代码：

```
int/* 这看起来并不像一个空格*/fox;
```

将变成：

```
int fox;
```

而且，实现可以用一个空格替换所有的空白字符序列（不包括换行符）。最后，程序已经准备好进入预处理阶段，预处理器查找一行中以#号开始的预处理指令。

16.2　明示常量：#define

#define 预处理器指令和其他预处理器指令一样，以#号作为一行的开始。ANSI 和后来的标准都允许#号前面有空格或制表符，而且还允许在#和指令的其余部分之间有空格。但是旧版本的 C 要求指令从一行最左边开始，而且#和指令其余部分之间不能有空格。指令可以出现在源文件的任何地方，其定义从指令出现的地方到该文件末尾有效。我们大量使用#define 指令来定义**明示常量**（*manifest constant*）（也叫做符号常量），但是该指令还有许多其他用途。程序清单 16.1 演示了#define 指令的一些用法和属性。

预处理器指令从#开始运行，到后面的第 1 个换行符为止。也就是说，指令的长度仅限于一行。然而，前面提到过，在预处理开始前，编译器会把多行物理行处理为一行逻辑行。

程序清单 16.1　preproc.c 程序

```
/* preproc.c -- 简单的预处理示例 */
#include <stdio.h>
#define TWO 2        /* 可以使用注释 */
#define OW "Consistency is the last refuge of the unimagina\
tive. - Oscar Wilde" /* 反斜杠把该定义延续到下一行 */

#define FOUR  TWO*TWO
#define PX printf("X is %d.\n", x)
#define FMT  "X is %d.\n"

int main(void)
{
    int x = TWO;

    PX;
    x = FOUR;
    printf(FMT, x);
    printf("%s\n", OW);
    printf("TWO: OW\n");

    return 0;
}
```

每行#define（逻辑行）都由 3 部分组成。第 1 部分是#define 指令本身。第 2 部分是选定的缩写，也称为**宏**。有些宏代表值（如本例），这些宏被称为**类对象宏**（*object-like macro*）。C 语言还有**类函数宏**（*function-like macro*），稍后讨论。宏的名称中不允许有空格，而且必须遵循 C 变量的命名规则：只能使用字符、数字和下划线（_）字符，而且首字符不能是数字。第 3 部分（指令行的其余部分）称为**替换列表**或**替换体**（见图 16.1）。一旦预处理器在程序中找到宏的示实例后，就会用替换体代替该宏（也有例外，稍后解释）。从宏变成最终替换文本的过程称为**宏展开**（*macro expansion*）。注意，可以在#define 行使用标准 C 注释。如前所述，每条注释都会被一个空格代替。

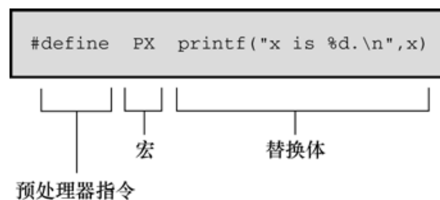


图 16.1　类对象宏定义的组成

运行该程序示例后，输出如下：

```
X is 2.
X is 4.
Consistency is the last refuge of the unimaginative. - Oscar Wilde
TWO: OW
```

下面分析具体的过程。下面的语句：

```
int x = TWO;
```

变成了：

```
int x = 2;
```

2 代替了 TWO。而语句：

```
PX;
```

变成了：

```
printf("X is %d.\n", x);
```

这里同样进行了替换。这是一个新用法，到目前为止我们只是用宏来表示明示常量。从该例中可以看出，宏可以表示任何字符串，甚至可以表示整个 C 表达式。但是要注意，虽然 PX 是一个字符串常量，它只打印一个名为 x 的变量。

下一行也是一个新用法。读者可能认为 FOUR 被替换成 4，但是实际的过程是：

```
x = FOUR;
```

变成了：

```
x = TWO*TWO;
```

即是：

```
x = 2*2;
```

宏展开到此处为止。由于编译器在编译期对所有的常量表达式（只包含常量的表达式）求值，所以预处理器不会进行实际的乘法运算，这一过程在编译时进行。预处理器不做计算，不对表达式求值，它只进行替换。

注意，宏定义还可以包含其他宏（一些编译器不支持这种嵌套功能）。

程序中的下一行：

```
printf (FMT, x);
```

变成了：

```
printf("X is %d.\n",x);
```

相应的字符串替换了 FMT。如果要多次使用某个冗长的字符串，这种方法比较方便。另外，也可以用下面的方法：

```
const char * fmt = "X is %d.\n";
```

然后可以把 fmt 作为 printf()的格式字符串。

下一行中，用相应的字符串替换 OW。双引号使替换的字符串成为字符串常量。编译器把该字符串储存在以空字符结尾的数组中。因此，下面的指令定义了一个字符常量：

```
#define HAL 'Z'
```

而下面的指令则定义了一个字符串（Z\0）：

```
#define HAP "Z"
```

在程序示例 16.1 中，我们在一行的结尾加一个反斜杠字符使该行扩展至下一行：

```
#define OW "Consistency is the last refuge of the unimagina\
tive. - Oscar Wilde"
```

注意，第 2 行要与第 1 行左对齐。如果这样做：

```
#define OW "Consistency is the last refuge of the unimagina\
    tive. - Oscar Wilde"
```

那么输出的内容是：

```
Consistency is the last refuge of the unimagina tive. - Oscar Wilde
```

第 2 行开始到 tive 之间的空格也算是字符串的一部分。

一般而言，预处理器发现程序中的宏后，会用宏等价的替换文本进行替换。如果替换的字符串中还包含宏，则继续替换这些宏。唯一例外的是双引号中的宏。因此，下面的语句：

```
printf("TWO: OW");
```

打印的是 TWO: OW，而不是打印：

```
2: Consistency is the last refuge of the unimaginative. - Oscar Wilde
```

要打印这行，应该这样写：

```
printf("%d: %s\n", TWO, OW);
```

这行代码中，宏不在双引号内。

那么，何时使用字符常量？对于绝大部分数字常量，应该使用字符常量。如果在算式中用字符常量代替数字，常量名能更清楚地表达该数字的含义。如果是表示数组大小的数字，用符号常量后更容易改变数组的大小和循环次数。如果数字是系统代码（如，EOF），用符号常量表示的代码更容易移植（只需改变 EOF 的定义）。助记、易更改、可移植，这些都是符号常量很有价值的特性。

C 语言现在也支持 const 关键字，提供了更灵活的方法。用 const 可以创建在程序运行过程中不能改变的变量，可具有文件作用域或块作用域。另一方面，宏常量可用于指定标准数组的大小和 const 变量的初始值。

```
#define LIMIT 20
const int LIM = 50;
static int data1[LIMIT];       // 有效
static int data2[LIM];         // 无效
const int LIM2 = 2 * LIMIT;    // 有效
const int LIM3 = 2 * LIM;      // 无效
```

这里解释一下上面代码中的“无效”注释。在 C 中，非自动数组的大小应该是整型常量表达式，这意味着表示数组大小的必须是整型常量的组合（如 5）、枚举常量和 sizeof 表达式，不包括 const 声明的值（这也是 C++和 C 的区别之一，在 C++中可以把 const 值作为常量表达式的一部分）。但是，有的实现可能接受其他形式的常量表达式。例如，GCC 4.7.3 不允许 data2 的声明，但是 Clang 4.6 允许。

16.2.1　记号

从技术角度来看，可以把宏的替换体看作是**记号**（*token*）型字符串，而不是字符型字符串。C 预处理器记号是宏定义的替换体中单独的“词”。用空白把这些词分开。例如：

```
#define FOUR 2*2
```

该宏定义有一个记号：2*2 序列。但是，下面的宏定义中：

```
#define SIX 2 * 3
```

有 3 个记号：2、*、3。

替换体中有多个空格时，字符型字符串和记号型字符串的处理方式不同。考虑下面的定义：

```
#define EIGHT 4 * 8
```

如果预处理器把该替换体解释为字符型字符串，将用 4 * 8 替换 EIGHT。即，额外的空格是替换体的一部分。如果预处理器把该替换体解释为记号型字符串，则用 3 个的记号 4 * 8（分别由单个空格分隔）来替换 EIGHT。换而言之，解释为字符型字符串，把空格视为替换体的一部分；解释为记号型字符串，把空格视为替换体中各记号的分隔符。在实际应用中，一些 C 编译器把宏替换体视为字符串而不是记号。在比这个例子更复杂的情况下，两者的区别才有实际意义。

顺带一提，C 编译器处理记号的方式比预处理器复杂。由于编译器理解 C 语言的规则，所以不要求代码中用空格来分隔记号。例如，C 编译器可以把 2*2 直接视为 3 个记号，因为它可以识别 2 是常量，*是运算符。

16.2.2　重定义常量

假设先把 LIMIT 定义为 20，稍后在该文件中又把它定义为 25。这个过程称为**重定义常量**。不同的实现采用不同的重定义方案。除非新定义与旧定义相同，否则有些实现会将其视为错误。另外一些实现允许重定义，但会给出警告。ANSI 标准采用第 1 种方案，只有新定义和旧定义完全相同才允许重定义。

具有相同的定义意味着替换体中的记号必须相同，且顺序也相同。因此，下面两个定义相同：

```
#define SIX 2 * 3
#define SIX 2    *    3
```

这两条定义都有 3 个相同的记号，额外的空格不算替换体的一部分。而下面的定义则与上面两条宏定义不同：

```
#define SIX 2*3
```

这条宏定义中只有一个记号，因此与前两条定义不同。如果需要重定义宏，使用#undef 指令（稍后讨论）。

如果确实需要重定义常量，使用 const 关键字和作用域规则更容易些。

16.3　在#define 中使用参数

在#define 中使用参数可以创建外形和作用与函数类似的**类函数宏**。带有参数的宏看上去很像函数，因为这样的宏也使用圆括号。类函数宏定义的圆括号中可以有一个或多个参数，随后这些参数出现在替换

体中，如图 16.2 所示。

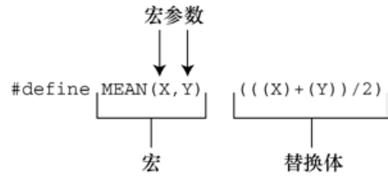


图 16.2　函数宏定义的组成

下面是一个类函数宏的示例：

```
#define SQUARE(X) X*X
```

在程序中可以这样用：

```
z = SQUARE(2);
```

这看上去像函数调用，但是它的行为和函数调用完全不同。程序清单 16.2 演示了类函数宏和另一个宏的用法。该示例中有一些陷阱，请读者仔细阅读序。

程序清单 16.2　mac_arg.c 程序

```
/* mac_arg.c -- 带参数的宏 */
#include <stdio.h>
#define SQUARE(X) X*X
#define PR(X)   printf("The result is %d.\n", X)
int main(void)
{
    int x = 5;
    int z;

    printf("x = %d\n", x);
    z = SQUARE(x);
    printf("Evaluating SQUARE(x): ");
    PR(z);
    z = SQUARE(2);
    printf("Evaluating SQUARE(2): ");
    PR(z);
    printf("Evaluating SQUARE(x+2): ");
    PR(SQUARE(x + 2));
    printf("Evaluating 100/SQUARE(2): ");
    PR(100 / SQUARE(2));
    printf("x is %d.\n", x);
    printf("Evaluating SQUARE(++x): ");
    PR(SQUARE(++x));
    printf("After incrementing, x is %x.\n", x);

    return 0;
}
```

SQUARE 宏的定义如下：

```
#define SQUARE(X) X*X
```

这里，SQUARE 是宏标识符，SQUARE(X)中的 X 是宏参数，X*X 是替换列表。程序清单 16.2 中出现 SQUARE(X)的地方都会被 X*X 替换。这与前面的示例不同，使用该宏时，既可以用 X，也可以用其他符号。宏定义中的 X 由宏调用中的符号代替。因此，SQUARE(2)替换为 2*2，X 实际上起到参数的作用。

然而，稍后你将看到，宏参数与函数参数不完全相同。下面是程序的输出。注意有些内容可能与我们的预期不符。实际上，你的编译器输出甚至与下面的结果完全不同。

```
x = 5
Evaluating SQUARE(x): The result is 25.
Evaluating SQUARE(2): The result is 4.
 Evaluating SQUARE(x+2): The result is 17.
Evaluating 100/SQUARE(2): The result is 100.
x is 5.
Evaluating SQUARE(++x): The result is 42.
After incrementing, x is 7.
```

前两行与预期相符，但是接下来的结果有点奇怪。程序中设置 x 的值为 5，你可能认为 SQUARE(x+2)应该是 7*7，即 49。但是，输出的结果是 17，这不是一个平方值！导致这样结果的原因是，我们前面提到过，预处理器不做计算、不求值，只替换字符序列。预处理器把出现 x 的地方都替换成 x+2。因此，x*x 变成了 x+2*x+2。如果 x 为 5，那么该表达式的值为：

```
5+2*5+2 = 5 + 10 + 2 = 17
```

该例演示了函数调用和宏调用的重要区别。函数调用在程序运行时把参数的值传递给函数。宏调用在编译之前把参数记号传递给程序。这两个不同的过程发生在不同时期。是否可以修改宏定义让 SQUARE(x+2)得 36？当然可以，要多加几个圆括号：

```
#define SQUARE(x) (x)*(x)
```

现在 SQUARE(x+2)变成了(x+2)*(x+2)，在替换字符串中使用圆括号就得到符合预期的乘法运算。

但是，这并未解决所有的问题。下面的输出行：

```
100/SQUARE(2)
```

将变成：

```
100/2*2
```

根据优先级规则，从左往右对表达式求值：(100/2)*2，即 50*2，得 100。把 SQUARE(x)定义为下面的形式可以解决这种混乱：

```
#define SQUARE(x) (x*x)
```

这样修改定义后得 100/(2*2)，即 100/4，得 25。

要处理前面的两种情况，要这样定义：

```
#define SQUARE(x) ((x)*(x))
```

因此，必要时要使用足够多的圆括号来确保运算和结合的正确顺序。

尽管如此，这样做还是无法避免程序中最后一种情况的问题。SQUARE(++x)变成了++x*++x，递增了两次 x，一次在乘法运算之前，一次在乘法运算之后：

```
++x*++x = 6*7 = 42
```

由于标准并未对这类运算规定顺序，所以有些编译器得 7*6。而有些编译器可能在乘法运算之前已经递增了 x，所以 7*7 得 49。在 C 标准中，对该表达式求值的这种情况称为未定义行为。无论哪种情况，x 的开始值都是 5，虽然从代码上看只递增了一次，但是 x 的最终值是 7。

解决这个问题最简单的方法是，避免用++x 作为宏参数。一般而言，不要在宏中使用递增或递减运算符。但是，++x 可作为函数参数，因为编译器会对++x 求值得 5 后，再把 5 传递给函数。

16.3.1 用宏参数创建字符串：#运算符

下面是一个类函数宏：

```
#define PSQR(X) printf("The square of X is %d.\n", ((X)*(X)));
```

假设这样使用宏：

```
PSQR(8);
```

输出为：

```
The square of X is 64.
```

注意双引号字符串中的 X 被视为普通文本，而不是一个可被替换的记号。

C 允许在字符串中包含宏参数。在类函数宏的替换体中，#号作为一个预处理运算符，可以把记号转换成字符串。例如，如果 x 是一个宏形参，那么#x 就是转换为字符串"x"的形参名。这个过程称为**字符串化**（*stringizing*）。程序清单 16.3 演示了该过程的用法。

程序清单 16.3　subst.c 程序

```
/* subst.c -- 在字符串中替换 */
#include <stdio.h>
#define PSQR(x) printf("The square of " #x " is %d.\n",((x)*(x)))

int main(void)
{
    int y = 5;

    PSQR(y);
    PSQR(2 + 4);

    return 0;
}
```

该程序的输出如下：

```
The square of y is 25.
The square of 2 + 4 is 36.
```

调用第 1 个宏时，用"y"替换#x。调用第 2 个宏时，用"2 + 4"替换#x。ANSI C 字符串的串联特性将这些字符串与 printf()语句的其他字符串组合，生成最终的字符串。例如，第 1 次调用变成：

```
printf("The square of " "y" " is %d.\n",((y)*(y)));
```

然后，字符串串联功能将这 3 个相邻的字符串组合成一个字符串：

```
"The square of y is %d.\n"
```

16.3.2　预处理器黏合剂：##运算符

与#运算符类似，##运算符可用于类函数宏的替换部分。而且，##还可用于对象宏的替换部分。##运算符把两个记号组合成一个记号。例如，可以这样做：

```
#define XNAME(n) x ## n
```

然后，宏 XNAME(4)将展开为 x4。程序清单 16.4 演示了##作为记号粘合剂的用法。

程序清单 16.4　glue.c 程序

```
// glue.c -- 使用##运算符
#include <stdio.h>
#define XNAME(n) x ## n
#define PRINT_XN(n) printf("x" #n " = %d\n", x ## n);

int main(void)
```

```
{
    int XNAME(1) = 14;     // 变成 int x1 = 14;
     int XNAME(2) = 20;    // 变成 int x2 = 20;
    int x3 = 30;
    PRINT_XN(1);           // 变成 printf("x1 = %d\n", x1);
    PRINT_XN(2);           // 变成 printf("x2 = %d\n", x2);
    PRINT_XN(3);           // 变成 printf("x3 = %d\n", x3);
      return 0;
}
```

该程序的输出如下：

```
x1 = 14
x2 = 20
x3 = 30
```

注意，PRINT_XN()宏用#运算符组合字符串，##运算符把记号组合为一个新的标识符。

16.3.3 变参宏：...和__VA_ARGS__

一些函数（如 printf()）接受数量可变的参数。stdvar.h 头文件（本章后面介绍）提供了工具，让用户自定义带可变参数的函数。C99/C11 也对宏提供了这样的工具。虽然标准中未使用“可变”（*variadic*）这个词，但是它已成为描述这种工具的通用词（虽然，C 标准的索引添加了字符串化(*stringizing*)词条，但是，标准并未把固定参数的函数或宏称为固定函数和不变宏）。

通过把宏参数列表中最后的参数写成省略号（即，3 个点...）来实现这一功能。这样，预定义宏__VA_ARGS__可用在替换部分中，表明省略号代表什么。例如，下面的定义：

```
#define PR(...) printf(__VA_ARGS__)
```

假设稍后调用该宏：

```
PR("Howdy");
PR("weight = %d, shipping = $%.2f\n", wt, sp);
```

对于第 1 次调用，__VA_ARGS__展开为 1 个参数："Howdy"。

对于第 2 次调用，__VA_ARGS__展开为 3 个参数："weight = %d, shipping = $%.2f\n"、wt、sp。

因此，展开后的代码是：

```
printf("Howdy");
printf("weight = %d, shipping = $%.2f\n", wt, sp);
```

程序清单 16.5 演示了一个示例，该程序使用了字符串的串联功能和#运算符。

程序清单 16.5 variadic.c 程序

```
// variadic.c -- 变参宏
#include <stdio.h>
#include <math.h>
#define PR(X, ...) printf("Message " #X ": " __VA_ARGS__)

int main(void)
{
    double x = 48;
    double y;

    y = sqrt(x);
```

```
    PR(1, "x = %g\n", x);
    PR(2, "x = %.2f, y = %.4f\n", x, y);

    return 0;
}
```

第 1 个宏调用，X 的值是 1，所以#X 变成"1"。展开后成为：

```
print("Message " "1" ": " "x = %g\n", x);
```

然后，串联 4 个字符，把调用简化为：

```
print("Message 1: x = %g\n", x);
```

下面是该程序的输出：

```
Message 1: x = 48
Message 2: x = 48.00, y = 6.9282
```

记住，省略号只能代替最后的宏参数：

```
#define WRONG(X, ..., Y) #X #__VA_ARGS__ #y //不能这样做
```

16.4　宏和函数的选择

有些编程任务既可以用带参数的宏完成，也可以用函数完成。应该使用宏还是函数？这没有硬性规定，但是可以参考下面的情况。

使用宏比使用普通函数复杂一些，稍有不慎会产生奇怪的副作用。一些编译器规定宏只能定义成一行。不过，即使编译器没有这个限制，也应该这样做。

宏和函数的选择实际上是时间和空间的权衡。宏生成内联代码，即在程序中生成语句。如果调用 20 次宏，即在程序中插入 20 行代码。如果调用函数 20 次，程序中只有一份函数语句的副本，所以节省了空间。然而另一方面，程序的控制必须跳转至函数内，随后再返回主调程序，这显然比内联代码花费更多的时间。

宏的一个优点是，不用担心变量类型（这是因为宏处理的是字符串，而不是实际的值）。因此，只要能用 int 或 float 类型都可以使用 SQUARE(x)宏。

C99 提供了第 3 种可替换的方法——内联函数。本章后面将介绍。

对于简单的函数，程序员通常使用宏，如下所示：

```
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y))
#define ABS(X) ((X) < 0 ? -(X) : (X))
#define ISSIGN(X) ((X) == '+' || (X) == '-' ? 1 : 0)
```

（如果 x 是一个代数符号字符，最后一个宏的值为 1，即为真。）

要注意以下几点。

- 记住宏名中不允许有空格，但是在替换字符串中可以有空格。ANSI C 允许在参数列表中使用空格。
- 用圆括号把宏的参数和整个替换体括起来。这样能确保被括起来的部分在下面这样的表达式中正确地展开：

  ```
  forks = 2 * MAX(guests + 3, last);
  ```
- 用大写字母表示宏函数的名称。该惯例不如用大写字母表示宏常量应用广泛。但是，大写字母可以提醒程序员注意，宏可能产生的副作用。
- 如果打算使用宏来加快程序的运行速度，那么首先要确定使用宏和使用函数是否会导致较大差异。在程序中只使用一次的宏无法明显减少程序的运行时间。在嵌套循环中使用宏更有助于提高效率。

许多系统提供程序分析器以帮助程序员压缩程序中最耗时的部分。

假设你开发了一些方便的宏函数，是否每写一个新程序都要重写这些宏？如果使用#include 指令，就不用这样做了。

16.5　文件包含：#include

当预处理器发现#include 指令时，会查看后面的文件名并把文件的内容包含到当前文件中，即替换源文件中的#include 指令。这相当于把被包含文件的全部内容输入到源文件#include 指令所在的位置。#include 指令有两种形式：

```
#include <stdio.h>          ←文件名在尖括号中
#include "mystuff.h"        ←文件名在双引号中
```

在 UNIX 系统中，尖括号告诉预处理器在标准系统目录中查找该文件。双引号告诉预处理器首先在当前目录中（或文件名中指定的其他目录）查找该文件，如果未找到再查找标准系统目录：

```
#include <stdio.h>          ←查找系统目录
#include "hot.h"            ←查找当前工作目录
#include "/usr/biff/p.h"    ←查找/usr/biff 目录
```

集成开发环境（IDE）也有标准路径或系统头文件的路径。许多集成开发环境提供菜单选项，指定用尖括号时的查找路径。在 UNIX 中，使用双引号意味着先查找本地目录，但是具体查找哪个目录取决于编译器的设定。有些编译器会搜索源代码文件所在的目录，有些编译器则搜索当前的工作目录，还有些搜索项目文件所在的目录。

ANSI C 不为文件提供统一的目录模型，因为不同的计算机所用的系统不同。一般而言，命名文件的方法因系统而异，但是尖括号和双引号的规则与系统无关。

为什么要包含文件？因为编译器需要这些文件中的信息。例如，stdio.h 文件中通常包含 EOF、NULL、getchar()和 putchar()的定义。getchar()和 putchar()被定义为宏函数。此外，该文件中还包含 C 的其他 I/O 函数。

C 语言习惯用.h 后缀表示**头文件**，这些文件包含需要放在程序顶部的信息。头文件经常包含一些预处理器指令。有些头文件（如 stdio.h）由系统提供，当然你也可以创建自己的头文件。

包含一个大型头文件不一定显著增加程序的大小。在大部分情况下，头文件的内容是编译器生成最终代码时所需的信息，而不是添加到最终代码中的材料。

16.5.1　头文件示例

假设你开发了一个存放人名的结构，还编写了一些使用该结构的函数。可以把不同的声明放在头文件中。程序清单 16.6 演示了一个这样的例子。

程序清单 16.6　names_st.h 头文件

```
// names_st.h -- names_st 结构的头文件
// 常量
#include <string.h>
#define SLEN 32

// 结构声明
struct names_st
{
```

```
    char first[SLEN];
    char last[SLEN];
};

// 类型定义
typedef struct names_st names;

// 函数原型
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);
```

该头文件包含了一些头文件中常见的内容：`#define` 指令、结构声明、`typedef` 和函数原型。注意，这些内容是编译器在创建可执行代码时所需的信息，而不是可执行代码。为简单起见，这个特殊的头文件过于简单。通常，应该用`#ifndef` 和`#define` 防止多重包含头文件。我们稍后介绍这些内容。

可执行代码通常在源代码文件中，而不是在头文件中。例如，程序清单 16.7 中有头文件中函数原型的定义。该程序包含了 `names_st.h` 头文件，所以编译器知道 `names` 类型。

程序清单 16.7　name_st.c 源文件

```
// names_st.c -- 定义 names_st.h 中的函数
#include <stdio.h>
#include "names_st.h"     // 包含头文件

// 函数定义
void get_names(names * pn)
{
    printf("Please enter your first name: ");
    s_gets(pn->first, SLEN);

    printf("Please enter your last name: ");
    s_gets(pn->last, SLEN);
}

void show_names(const names * pn)
{
    printf("%s %s", pn->first, pn->last);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
```

```
                continue;      // 处理输入行中的剩余字符
        }
        return ret_val;
}
```

get_names()函数通过 s_gets()函数调用了 fgets()函数，避免了目标数组溢出。程序清单 16.8 使用了程序清单 16.6 的头文件和程序清单 16.7 的源文件。

程序清单 16.8　useheader.c 程序

```
// useheader.c -- 使用 names_st 结构
#include <stdio.h>
#include "names_st.h"
// 记住要链接 names_st.c

int main(void)
{
    names candidate;

    get_names(&candidate);
    printf("Let's welcome ");
    show_names(&candidate);
    printf(" to this program!\n");
    return 0;
}
```

下面是该程序的输出：

```
Please enter your first name: Ian
Please enter your last name: Smersh
Let's welcome Ian Smersh to this program!
```

该程序要注意下面几点。

- 两个源代码文件都使用 names_st 类型结构，所以它们都必须包含 names_st.h 头文件。
- 必须编译和链接 names_st.c 和 useheader.c 源代码文件。
- 声明和指令放在 nems_st.h 头文件中，函数定义放在 names_st.c 源代码文件中。

16.5.2　使用头文件

浏览任何一个标准头文件都可以了解头文件的基本信息。头文件中最常用的形式如下。

- **明示常量**——例如，stdio.h 中定义的 EOF、NULL 和 BUFSIZE（标准 I/O 缓冲区大小）。
- **宏函数**——例如，getc(stdin)通常用 getchar()定义，而 getc()经常用于定义较复杂的宏，头文件 ctype.h 通常包含 ctype 系列函数的宏定义。
- **函数声明**——例如，string.h 头文件（一些旧的系统中是 strings.h）包含字符串函数系列的函数声明。在 ANSI C 和后面的标准中，函数声明都是函数原型形式。
- **结构模版定义**——标准 I/O 函数使用 FILE 结构，该结构中包含了文件和与文件缓冲区相关的信息。FILE 结构在头文件 stdio.h 中。
- **类型定义**——标准 I/O 函数使用指向 FILE 的指针作为参数。通常，stdio.h 用#define 或 typedef 把 FILE 定义为指向结构的指针。类似地，size_t 和 time_t 类型也定义在头文件中。

许多程序员都在程序中使用自己开发的标准头文件。如果开发一系列相关的函数或结构，那么这种方法特别有价值。

另外，还可以使用头文件声明外部变量供其他文件共享。例如，如果已经开发了共享某个变量的一系列函数，该变量报告某种状况（如，错误情况），这种方法就很有效。这种情况下，可以在包含这些函数声明的源代码文件定义一个文件作用域的外部链接变量：

```
int status = 0;         // 该变量具有文件作用域，在源代码文件
```

然后，可以在与源代码文件相关联的头文件中进行引用式声明：

```
extern int status;      // 在头文件中
```

这行代码会出现在包含了该头文件的文件中，这样使用该系列函数的文件都能使用这个变量。虽然源代码文件中包含该头文件后也包含了该声明，但是只要声明的类型一致，在一个文件中同时使用定义式声明和引用式声明没问题。

需要包含头文件的另一种情况是，使用具有文件作用域、内部链接和 const 限定符的变量或数组。const 防止值被意外修改，static 意味着每个包含该头文件的文件都获得一份副本。因此，不需要在一个文件中进行定义式声明，在其他文件中进行引用式声明。

#include 和#define 指令是最常用的两个 C 预处理器特性。接下来，我们介绍一些其他指令。

16.6　其他指令

程序员可能要为不同的工作环境准备 C 程序和 C 库包。不同的环境可能使用不同的代码类型。预处理器提供一些指令，程序员通过修改#define 的值即可生成可移植的代码。#undef 指令取消之前的#define 定义。#if、#ifdef、#ifndef、#else、#elif 和#endif 指令用于指定什么情况下编写哪些代码。#line 指令用于重置行和文件信息，#error 指令用于给出错误消息，#pragma 指令用于向编译器发出指令。

16.6.1　#undef 指令

#undef 指令用于“取消”已定义的#define 指令。也就是说，假设有如下定义：

```
#define LIMIT 400
```

然后，下面的指令：

```
#undef LIMIT
```

将移除上面的定义。现在就可以把 LIMIT 重新定义为一个新值。即使原来没有定义 LIMIT，取消 LIMIT 的定义仍然有效。如果想使用一个名称，又不确定之前是否已经用过，为安全起见，可以用#undef 指令取消该名字的定义。

16.6.2　从 C 预处理器角度看已定义

处理器在识别标识符时，遵循与 C 相同的规则：标识符可以由大写字母、小写字母、数字和下划线字符组成，且首字符不能是数字。当预处理器在预处理器指令中发现一个标识符时，它会把该标识符当作**已定义的**或**未定义的**。这里的**已定义**表示由预处理器定义。如果标识符是同一个文件中由前面的#define 指令创建的宏名，而且没有用#undef 指令关闭，那么该标识符是已定义的。如果标识符不是宏，假设是一个文件作用域的 C 变量，那么该标识符对预处理器而言就是未定义的。

已定义宏可以是对象宏，包括空宏或类函数宏：

```
#define LIMIT 1000          // LIMIT 是已定义的
```

```
#define GOOD                 // GOOD 是已定义的
#define A(X) ((-(X))*(X))    // A 是已定义的
int q;                       // q 不是宏，因此是未定义的
#undef GOOD                  // GOOD 取消定义，是未定义的
```

注意，#define宏的作用域从它在文件中的声明处开始，直到用#undef指令取消宏为止，或延伸至文件尾（以二者中先满足的条件作为宏作用域的结束）。另外还要注意，如果宏通过头文件引入，那么#define在文件中的位置取决于#include指令的位置。

稍后将介绍几个预定义宏，如__DATE__和__FILE__。这些宏一定是已定义的，而且不能取消定义。

16.6.3　条件编译

可以使用其他指令创建**条件编译**（*conditinal compilation*）。也就是说，可以使用这些指令告诉编译器根据编译时的条件执行或忽略信息（或代码）块。

1. #ifdef、#else和#endif指令

我们用一个简短的示例来演示条件编译的情况。考虑下面的代码：

```
#ifdef MAVIS
    #include "horse.h"// 如果已经用#define定义了 MAVIS，则执行下面的指令
    #define STABLES 5
#else
    #include "cow.h"       //如果没有用#define定义 MAVIS，则执行下面的指令
    #define STABLES 15
#endif
```

这里使用的较新的编译器和 ANSI 标准支持的缩进格式。如果使用旧的编译器，必须左对齐所有的指令或至少左对齐#号，如下所示：

```
#ifdef MAVIS
#include "horse.h"         // 如果已经用#define定义了 MAVIS，则执行下面的指令
#define STABLES 5
#else
#include "cow.h"           //如果没有用#define定义 MAVIS，则执行下面的指令
#define STABLES 15
#endif
```

#ifdef指令说明，如果预处理器已定义了后面的标识符（MAVIS），则执行#else或#endif指令之前的所有指令并编译所有C代码（先出现哪个指令就执行到哪里）。如果预处理器未定义MAVIS，且有 #else指令，则执行#else和#endif指令之间的所有代码。

#ifdef　#else很像C的if　else。两者的主要区别是，预处理器不识别用于标记块的花括号（{}），因此它使用#else（如果需要）和#endif（必须存在）来标记指令块。这些指令结构可以嵌套。也可以用这些指令标记C语句块，如程序清单16.9所示。

程序清单16.9　ifdef.c程序

```
/* ifdef.c -- 使用条件编译 */
#include <stdio.h>
#define JUST_CHECKING
#define LIMIT 4

int main(void)
{
```

```
    int i;
    int total = 0;

    for (i = 1; i <= LIMIT; i++)
    {
        total += 2 * i*i + 1;
#ifdef JUST_CHECKING
        printf("i=%d, running total = %d\n", i, total);
#endif
    }
    printf("Grand total = %d\n", total);

    return 0;
}
```

编译并运行该程序后，输出如下：

```
i=1, running total = 3
i=2, running total = 12
i=3, running total = 31
i=4, running total = 64
Grand total = 64
```

如果省略 JUST_CHECKING 定义（把它放在 C 注释中，或者使用#undef 指令取消它的定义）并重新编译该程序，只会输出最后一行。可以用这种方法在调试程序。定义 JUST_CHECKING 并合理使用#ifdef，编译器将执行用于调试的程序代码，打印中间值。调试结束后，可移除 JUST_CHECKING 定义并重新编译。如果以后还需要使用这些信息，重新插入定义即可。这样做省去了再次输入额外打印语句的麻烦。#ifdef 还可用于根据不同的 C 实现选择合适的代码块。

2. #ifndef 指令

#ifndef 指令与#ifdef 指令的用法类似，也可以和#else、#endif 一起使用，但是它们的逻辑相反。#ifndef 指令判断后面的标识符是否是未定义的，常用于定义之前未定义的常量。如下所示：

```
/* arrays.h */
#ifndef SIZE
    #define SIZE 100
#endif
```

（旧的实现可能不允许使用缩进的#define）

通常，包含多个头文件时，其中的文件可能包含了相同宏定义。#ifndef 指令可以防止相同的宏被重复定义。在首次定义一个宏的头文件中用#ifndef 指令激活定义，随后在其他头文件中的定义都被忽略。

#ifndef 指令还有另一种用法。假设有上面的 arrays.h 头文件，然后把下面一行代码放入一个头文件中：

```
#include "arrays.h"
```

SIZE 被定义为 100。但是，如果把下面的代码放入该头文件：

```
#define SIZE 10
#include "arrays.h"
```

SIZE 则被设置为 10。这里，当执行到#include "arrays.h"这行，处理 array.h 中的代码时，由于 SIZE 是已定义的，所以跳过了#define SIZE 100 这行代码。鉴于此，可以利用这种方法，用一个较小的数组测试程序。测试完毕后，移除#define SIZE 10 并重新编译。这样，就不用修改头文件数组本身了。

#ifndef 指令通常用于防止多次包含一个文件。也就是说，应该像下面这样设置头文件：

```
/* things.h */
#ifndef THINGS_H_
    #define THINGS_H_
    /* 省略了头文件中的其他内容*/
#endif
```

假设该文件被包含了多次。当预处理器首次发现该文件被包含时，THINGS_H_是未定义的，所以定义了 THINGS_H_，并接着处理该文件的其他部分。当预处理器第 2 次发现该文件被包含时，THINGS_H_是已定义的，所以预处理器跳过了该文件的其他部分。

为何要多次包含一个文件？最常见的原因是，许多被包含的文件中都包含着其他文件，所以显式包含的文件中可能包含着已经包含的其他文件。这有什么问题？在被包含的文件中有某些项（如，一些结构类型的声明）只能在一个文件中出现一次。C 标准头文件使用#ifndef 技巧避免重复包含。但是，这存在一个问题：如何确保待测试的标识符没有在别处定义。通常，实现的供应商使用这些方法解决这个问题：用文件名作为标识符、使用大写字母、用下划线字符代替文件名中的点字符、用下划线字符做前缀或后缀（可能使用两条下划线）。例如，查看 stdio.h 头文件，可以发现许多类似的代码：

```
#ifndef _STDIO_H
#define _STDIO_H
// 省略了文件的内容
#endif
```

你也可以这样做。但是，由于标准保留使用下划线作为前缀，所以在自己的代码中不要这样写，避免与标准头文件中的宏发生冲突。程序清单 16.10 修改了程序清单 16.6 中的头文件，使用#ifndef 避免文件被重复包含。

程序清单 16.10　names.c 程序

```
// names.h --修订后的 names_st 头文件，避免重复包含

#ifndef NAMES_H_
#define NAMES_H_

// 明示常量
#define SLEN 32

// 结构声明
struct names_st
{
    char first[SLEN];
    char last[SLEN];
};

// 类型定义
typedef struct names_st names;

// 函数原型
void get_names(names *);
void show_names(const names *);
char * s_gets(char * st, int n);

#endif
```

用程序清单 16.11 的程序测试该头文件没问题，但是如果把清单 16.10 中的#ifndef 保护删除后，程序就无法通过编译。

程序清单 16.11 doubincl.c 程序

```
// doubincl.c -- 包含头文件两次
#include <stdio.h>
#include "names.h"
#include "names.h"   // 不小心第 2 次包含头文件

int main()
{
    names winner = { "Less", "Ismoor" };
    printf("The winner is %s %s.\n", winner.first,
        winner.last);
    return 0;
}
```

3. #if 和#elif 指令

#if 指令很像 C 语言中的 if。#if 后面跟整型常量表达式，如果表达式为非零，则表达式为真。可以在指令中使用 C 的关系运算符和逻辑运算符：

```
#if SYS == 1
#include "ibm.h"
#endif
```

可以按照 if else 的形式使用#elif（早期的实现不支持#elif）。例如，可以这样写：

```
#if SYS == 1
    #include "ibmpc.h"
#elif SYS == 2
    #include "vax.h"
#elif SYS == 3
    #include "mac.h"
#else
    #include "general.h"
#endif
```

较新的编译器提供另一种方法测试名称是否已定义，即用#if defined (VAX)代替#ifdef VAX。

这里，defined 是一个预处理运算符，如果它的参数是用#defined 定义过，则返回 1；否则返回 0。这种新方法的优点是，它可以和#elif 一起使用。下面用这种形式重写前面的示例：

```
#if defined (IBMPC)
    #include "ibmpc.h"
#elif defined (VAX)
    #include "vax.h"
#elif defined (MAC)
    #include "mac.h"
#else
    #include "general.h"
#endif
```

如果在 VAX 机上运行这几行代码，那么应该在文件前面用下面的代码定义 VAX：

```
#define VAX
```

条件编译还有一个用途是让程序更容易移植。改变文件开头部分的几个关键的定义，即可根据不同的系统设置不同的值和包含不同的文件。

16.6.4　预定义宏

C 标准规定了一些预定义宏，如表 16.1 所列。

表 16.1　预 定 义 宏

宏	含义
__DATE__	预处理的日期（"Mmm dd yyyy"形式的字符串字面量，如 Nov 23 2013）
__FILE__	表示当前源代码文件名的字符串字面量
__LINE__	表示当前源代码文件中行号的整型常量
__STDC__	设置为 1 时，表明实现遵循 C 标准
__STDC_HOSTED__	本机环境设置为 1；否则设置为 0
__STDC_VERSION__	支持 C99 标准，设置为 199901L；支持 C11 标准，设置为 201112L
__TIME__	翻译代码的时间，格式为“ hh:mm:ss ”

C99 标准提供一个名为__func__的预定义标识符，它展开为一个代表函数名的字符串（该函数包含该标识符）。那么，__func__必须具有函数作用域，而从本质上看宏具有文件作用域。因此，__func__是 C 语言的预定义标识符，而不是预定义宏。

程序清单 16.12 中使用了一些预定义宏和预定义标识符。注意，其中一些是 C99 新增的，所以不支持 C99 的编译器可能无法识别它们。如果使用 GCC，必须设置-std=c99 或-std=c11。

程序清单 16.12　predef.c 程序

```
// predef.c -- 预定义宏和预定义标识符
#include <stdio.h>
void why_me();

int main()
{
    printf("The file is %s.\n", __FILE__);
    printf("The date is %s.\n", __DATE__);
    printf("The time is %s.\n", __TIME__);
    printf("The version is %ld.\n", __STDC_VERSION__);
    printf("This is line %d.\n", __LINE__);
    printf("This function is %s\n", __func__);
    why_me();

    return 0;
}

void why_me()
{
    printf("This function is %s\n", __func__);
    printf("This is line %d.\n", __LINE__);
}
```

下面是该程序的输出：

```
The file is predef.c.
The date is Sep 23 2013.
The time is 22:01:09.
```

```
The version is 201112.
This is line 11.
This function is main
This function is why_me
This is line 21.
```

16.6.5　#line 和#error

#line 指令重置__LINE__和__FILE__宏报告的行号和文件名。可以这样使用#line：

```
#line 1000             // 把当前行号重置为 1000
#line 10 "cool.c"      // 把行号重置为 10，把文件名重置为 cool.c
```

#error 指令让预处理器发出一条错误消息，该消息包含指令中的文本。如果可能的话，编译过程应该中断。可以这样使用#error 指令：

```
#if __STDC_VERSION__ != 201112L
#error Not C11

#endif
```

编译以上代码生成后，输出如下：

```
$ gcc newish.c
newish.c:14:2: error: #error Not C11
$ gcc -std=c11 newish.c
$
```

如果编译器只支持旧标准，则会编译失败，如果支持 C11 标准，就能成功编译。

16.6.6　#pragma

在现在的编译器中，可以通过命令行参数或 IDE 菜单修改编译器的一些设置。#pragma 把编译器指令放入源代码中。例如，在开发 C99 时，标准被称为 C9X，可以使用下面的**编译指示**（*pragma*）让编译器支持 C9X：

```
#pragma c9x on
```

一般而言，编译器都有自己的编译指示集。例如，编译指示可能用于控制分配给自动变量的内存量，或者设置错误检查的严格程度，或者启用非标准语言特性等。C99 标准提供了 3 个标准编译指示，但是超出了本书讨论的范围。

C99 还提供_Pragma 预处理器运算符，该运算符把字符串转换成普通的编译指示。例如：

```
_Pragma("nonstandardtreatmenttypeB on")
```

等价于下面的指令：

```
#pragma nonstandardtreatmenttypeB on
```

由于该运算符不使用#符号，所以可以把它作为宏展开的一部分：

```
#define PRAGMA(X) _Pragma(#X)
#define LIMRG(X) PRAGMA(STDC CX_LIMITED_RANGE X)
```

然后，可以使用类似下面的代码：

```
LIMRG ( ON )
```

顺带一提，下面的定义看上去没问题，但实际上无法正常运行：

```
#define LIMRG(X) _Pragma(STDC CX_LIMITED_RANGE #X)
```

问题在于这行代码依赖字符串的串联功能，而预处理过程完成之后才会串联字符串。

_Pragma 运算符完成“解字符串”（*destringizing*）的工作，即把字符串中的转义序列转换成它所代表

的字符。因此，

```
_Pragma("use_bool \"true \"false")
```

变成了：

```
#pragma use_bool "true "false
```

16.6.7 泛型选择（C11）

在程序设计中，**泛型编程**（*generic programming*）指那些没有特定类型，但是一旦指定一种类型，就可以转换成指定类型的代码。例如，C++在模板中可以创建泛型算法，然后编译器根据指定的类型自动使用实例化代码。C没有这种功能。然而，C11新增了一种表达式，叫作**泛型选择表达式**（*generic selection expression*），可根据表达式的类型（即表达式的类型是 int、double 还是其他类型）选择一个值。泛型选择表达式不是预处理器指令，但是在一些泛型编程中它常用作#define 宏定义的一部分。

下面是一个泛型选择表达式的示例：

```
_Generic(x, int: 0, float: 1, double: 2, default: 3)
```

_Generic 是 C11 的关键字。_Generic 后面的圆括号中包含多个用逗号分隔的项。第 1 个项是一个表达式，后面的每个项都由一个类型、一个冒号和一个值组成，如 float: 1。第 1 个项的类型匹配哪个标签，整个表达式的值是该标签后面的值。例如，假设上面表达式中 x 是 int 类型的变量，x 的类型匹配 int:标签，那么整个表达式的值就是 0。如果没有与类型匹配的标签，表达式的值就是 default:标签后面的值。泛型选择语句与 switch 语句类似，只是前者用表达式的类型匹配标签，而后者用表达式的值匹配标签。

下面是一个把泛型选择语句和宏定义组合的例子：

```
#define MYTYPE(X) _Generic((X),\
    int: "int",\
    float : "float",\
    double: "double",\
    default: "other"\
)
```

宏必须定义为一条逻辑行，但是可以用\把一条逻辑行分隔成多条物理行。在这种情况下，对泛型选择表达式求值得字符串。例如，对 MYTYPE(5)求值得"int"，因为值 5 的类型与 int:标签匹配。程序清单 16.13 演示了这种用法。

程序清单 16.13 mytype.c 程序

```
// mytype.c

#include <stdio.h>

#define MYTYPE(X) _Generic((X),\
    int: "int",\
    float : "float",\
    double: "double",\
    default: "other"\
)

int main(void)
{
    int d = 5;

    printf("%s\n", MYTYPE(d));      // d 是 int 类型
    printf("%s\n", MYTYPE(2.0*d));  // 2.0 * d 是 double 类型
```

```
    printf("%s\n", MYTYPE(3L));     // 3L 是 long 类型
    printf("%s\n", MYTYPE(&d));     // &d 的类型是 int *
    return 0;
}
```

下面是该程序的输出：

```
int
double
other
other
```

MYTYPE()最后两个示例所用的类型与标签不匹配，所以打印默认的字符串。可以使用更多类型标签来扩展宏的能力，但是该程序主要是为了演示_Generic 的基本工作原理。

对一个泛型选择表达式求值时，程序不会先对第一个项求值，它只确定类型。只有匹配标签的类型后才会对表达式求值。

可以像使用独立类型（“泛型”）函数那样使用_Generic 定义宏。本章后面介绍 math 库时会给出一个示例。

16.7　内联函数（C99）

通常，函数调用都有一定的开销，因为函数的调用过程包括建立调用、传递参数、跳转到函数代码并返回。使用宏使代码内联，可以避免这样的开销。C99 还提供另一种方法：**内联函数**（*inline function*）。读者可能顾名思义地认为内联函数会用内联代码替换函数调用。其实 C99 和 C11 标准中叙述的是：“把函数变成内联函数建议尽可能快地调用该函数，其具体效果由实现定义”。因此，把函数变成内联函数，编译器可能会用内联代码替换函数调用，并（或）执行一些其他的优化，但是也可能不起作用。

创建内联函数的定义有多种方法。标准规定具有内部链接的函数可以成为内联函数，还规定了内联函数的定义与调用该函数的代码必须在同一个文件中。因此，最简单的方法是使用函数说明符 inline 和存储类别说明符 static。通常，内联函数应定义在首次使用它的文件中，所以内联函数也相当于函数原型。如下所示：

```
#include <stdio.h>
inline static void eatline()    // 内联函数定义/原型
{
    while (getchar() != '\n')
        continue;
}

int main()
{
    ...
    eatline();              // 函数调用
    ...
}
```

编译器查看内联函数的定义（也是原型），可能会用函数体中的代码替换 eatline()函数调用。也就是说，效果相当于在函数调用的位置输入函数体中的代码：

```
#include <stdio.h>
inline static void eatline() //内联函数定义/原型
{
    while (getchar() != '\n')
```

```
        continue;
}

int main()
{
    ...
    while (getchar() != '\n') //替换函数调用
        continue;
    ...
}
```

由于并未给内联函数预留单独的代码块，所以无法获得内联函数的地址（实际上可以获得地址，不过这样做之后，编译器会生成一个非内联函数）。另外，内联函数无法在调试器中显示。

内联函数应该比较短小。把较长的函数变成内联并未节约多少时间，因为执行函数体的时间比调用函数的时间长得多。

编译器优化内联函数必须知道该函数定义的内容。这意味着内联函数定义与函数调用必须在同一个文件中。鉴于此，一般情况下内联函数都具有内部链接。因此，如果程序有多个文件都要使用某个内联函数，那么这些文件中都必须包含该内联函数的定义。最简单的做法是，把内联函数定义放入头文件，并在使用该内联函数的文件中包含该头文件即可。

```
// eatline.h
#ifndef EATLINE_H_
#define EATLINE_H_
inline static void eatline()
{
    while (getchar() != '\n')
        continue;
}
#endif
```

一般都不在头文件中放置可执行代码，内联函数是个特例。因为内联函数具有内部链接，所以在多个文件中定义同一个内联函数不会产生什么问题。

与 C++不同的是，C 还允许混合使用内联函数定义和外部函数定义（具有外部链接的函数定义）。例如，一个程序中使用下面 3 个文件：

```
//file1.c
...
inline static double square(double);
double square(double x) { return x * x; }
int main()
{
    double q = square(1.3);
    ...

//file2.c
...
double square(double x) { return (int) (x*x); }
void spam(double v)
{
    double kv = square(v);
    ...

//file3.c
...
```

```
inline double square(double x) { return (int) (x * x + 0.5); }
void masp(double w)
{
    double kw = square(w);
    ...
```

如上述代码所示，3 个文件中都定义了 square()函数。file1.c 文件中是 inline static 定义；file2.c 文件中是普通的函数定义（因此具有外部链接）；file3.c 文件中是 inline 定义，省略了 static。

3 个文件中的函数都调用了 square()函数，这会发生什么情况？。file1.c 文件中的 main()使用 square()的局部 static 定义。由于该定义也是 inline 定义，所以编译器有可能优化代码，也许会内联该函数。file2.c 文件中，spam()函数使用该文件中 square()函数的定义，该定义具有外部链接，其他文件也可见。file3.c 文件中，编译器既可以使用该文件中 square()函数的内联定义，也可以使用 file2.c 文件中的外部链接定义。如果像 file3.c 那样，省略 file1.c 文件 inline 定义中的 static，那么该 inline 定义被视为可替换的外部定义。

注意 GCC 在 C99 之前就使用一些不同的规则实现了内联函数，所以 GCC 可以根据当前编译器的标记来解释 inline。

16.8　_Noreturn 函数（C11）

C99 新增 inline 关键字时，它是唯一的函数说明符（关键字 extern 和 static 是存储类别说明符，可应用于数据对象和函数）。C11 新增了第 2 个函数说明符_Noreturn，表明调用完成后函数不返回主调函数。exit()函数是_Noreturn 函数的一个示例，一旦调用 exit()，它不会再返回主调函数。注意，这与 void 返回类型不同。void 类型的函数在执行完毕后返回主调函数，只是它不提供返回值。

_Noreturn 的目的是告诉用户和编译器，这个特殊的函数不会把控制返回主调程序。告诉用户以免滥用该函数，通知编译器可优化一些代码。

16.9　C 库

最初，并没有官方的 C 库。后来，基于 UNIX 的 C 实现成为了标准。ANSI C 委员会主要以这个标准为基础，开发了一个官方的标准库。在意识到 C 语言的应用范围不断扩大后，该委员会重新定义了这个库，使之可以应用于其他系统。

我们讨论过一些标准库中的 I/O 函数、字符函数和字符串函数。本章将介绍更多函数。不过，首先要学习如何使用库。

16.9.1　访问 C 库

如何访问 C 库取决于实现，因此你要了解当前系统的一般情况。首先，可以在多个不同的位置找到库函数。例如，getchar()函数通常作为宏定义在 stdio.h 头文件中，而 strlen()通常在库文件中。其次，不同的系统搜索这些函数的方法不同。下面介绍 3 种可能的方法。

1. 自动访问

在一些系统中，只需编译程序，就可使用一些常用的库函数。

记住，在使用函数之前必须先声明函数的类型，通过包含合适的头文件即可完成。在描述库函数的用

户手册中，会指出使用某函数时应包含哪个头文件。但是在一些旧系统上，可能必须自己输入函数声明。再次提醒读者，用户手册中指明了函数类型。另外，附录 B“参考资料”中根据头文件分组，总结了 ANSI C 库函数。

过去，不同的实现使用的头文件名不同。ANSI C 标准把库函数分为多个系列，每个系列的函数原型都放在一个特定的头文件中。

2. 文件包含

如果函数被定义为宏，那么可以通过`#include` 指令包含定义宏函数的文件。通常，类似的宏都放在合适名称的头文件中。例如，许多系统（包括所有的 ANSI C 系统）都有 `ctype.h` 文件，该文件中包含了一些确定字符性质（如大写、数字等）的宏。

3. 库包含

在编译或链接程序的某些阶段，可能需要指定库选项。即使在自动检查标准库的系统中，也会有不常用的函数库。必须通过编译时选项显式指定这些库。注意，这个过程与包含头文件不同。头文件提供函数声明或原型，而库选项告诉系统到哪里查找函数代码。虽然这里无法涉及所有系统的细节，但是可以提醒读者应该注意什么。

16.9.2 使用库描述

篇幅有限，我们无法讨论完整的库。但是，可以看几个具有代表性的示例。首先，了解函数文档。

可以在多个地方找到函数文档。你所使用的系统可能有在线手册，集成开发环境通常都有在线帮助。C 实现的供应商可能提供描述库函数的纸质版用户手册，或者把这些材料放在 CD-ROM 中或网上。有些出版社也出版 C 库函数的参考手册。这些材料中，有些是一般材料，有些则是针对特定实现的。本书附录 B 中提供了一个库函数的总结。

阅读文档的关键是看懂函数头。许多内容随时间变化而变化。下面是旧的 UNIX 文档中，关于 `fread()` 的描述：

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

首先，给出了应该包含的文件，但是没有给出 `fread()`、`ptr`、`sizeof(*ptr)`或 `nitems` 的类型。过去，默认类型都是 `int`，但是从描述中可以看出 `ptr` 是一个指针（在早期的 C 中，指针被作为整数处理）。参数 `stream` 声明为指向 `FILE` 的指针。上面的函数声明中的第 2 个参数看上去像是 `sizeof` 运算符，而实际上这个参数的值应该是 `ptr` 所指向对象的大小。虽然用 `sizeof` 作为参数没什么问题，但是用 `int` 类型的值作为参数更符合语法。

后来，上面的描述变成了：

```
#include <stdio.h>

int fread(ptr, size, nitems, stream;)
char *ptr;
int size, nitems;
FILE *stream;
```

现在，所有的类型都显式说明，`ptr` 作为指向 `char` 的指针。

ANSI C90 标准提供了下面的描述：

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

首先，使用了新的函数原型格式。其次，改变了一些类型。size_t 类型被定义为 sizeof 运算符的返回值类型——无符号整数类型，通常是 unsigned int 或 unsigned long。stddef.h 文件中包含了 size_t 类型的 typedef 或#define 定义。其他文件（包括 stdio.h）通过包含 stddef.h 来包含这个定义。许多函数（包括 fread()）的实际参数中都要使用 sizeof 运算符，形式参数的 size_t 类型中正好匹配这种常见的情况。

另外，ANSI C 把指向 void 的指针作为一种通用指针，用于指针指向不同类型的情况。例如，fread()的第 1 个参数可能是指向一个 double 类型数组的指针，也可能是指向其他类型结构的指针。如果假设实际参数是一个指向内含 20 个 double 类型元素数组的指针，且形式参数是指向 void 的指针，那么编译器会选用合适的类型，不会出现类型冲突的问题。

C99/C11 标准在以上的描述中加入了新的关键字 restric：

```
#include <stdio.h>
size_t fread(void * restrict ptr, size_t size,size_t nmemb, FILE * restrict stream);
```

接下来，我们讨论一些特殊的函数。

16.10　数学库

数学库中包含许多有用的数学函数。math.h 头文件提供这些函数的原型。表 16.2 中列出了一些声明在 math.h 中的函数。注意，函数中涉及的角度都以弧度为单位（1 弧度=180/π=57.296 度）。参考资料 V“新增 C99 和 C11 标准的 ANSI C 库”列出了 C99 和 C11 标准的所有函数。

表 16.2　ANSI C 标准的一些数学函数

原型	描述
double acos(double x)	返回余弦值为 x 的角度（0～π 弧度）
double asin(double x)	返回正弦值为 x 的角度（−π/2～π/2 弧度）
double atan(double x)	返回正切值为 x 的角度（−π/2～π/2 弧度）
double atan2(double y,double x)	返回正弦值为 y/x 的角度（−π～π 弧度）
double cos(double x)	返回 x 的余弦值，x 的单位为弧度
double sin(double x)	返回 x 的正弦值，x 的单位为弧度
double tan(double x)	返回 x 的正切值，x 的单位为弧度
double exp(double x)	返回 x 的指数函数的值（e^x）
double log(double x)	返回 x 的自然对数值
double log10(double x)	返回 x 的以 10 为底的对数值
double pow(double x, doubley)	返回 x 的 y 次幂
double sqrt(double x)	返回 x 的平方值
double cbrt(double x)	返回 x 的立方值
double ceil(double x)	返回不小于 x 的最小整数值
double fabs(double x)	返回 x 的绝对值
double floor(double x)	返回不大于 x 的最大整数值

16.10.1　三角问题

我们可以使用数学库解决一些常见的问题：把 x/y 坐标转换为长度和角度。例如，在网格上画了一条线，该线条水平穿过了 4 个单元（x 的值），垂直穿过了 3 个单元（y 的值）。那么，该线的长度（量）和方向是什么？根据数学的三角公式可知：

大小 =square root (x^2+y^2)

角度 = arctan(y/x)

数学库提供平方根函数和一对反正切函数，所以可以用 C 程序表示这个问题。平方根函数是 `sqrt()`，接受一个 `double` 类型的参数，并返回参数的平方根，也是 `double` 类型。

`atan()`函数接受一个 `double` 类型的参数（即正切值），并返回一个角度（该角度的正切值就是参数值）。但是，当线的 x 值和 y 值均为-5 时，`atan()`函数产生混乱。因为`(-5)/(-5)`得 `1`，所以 `atan()`返回 45°，该值与 x 和 y 均为 5 时的返回值相同。也就是说，`atan()`无法区分角度相同但反向相反的线（实际上，`atan()`返回值的单位是弧度而不是度，稍后介绍两者的转换）。

当然，C 库还提供了 `atan2()`函数。它接受两个参数：x 的值和 y 的值。这样，通过检查 x 和 y 的正负号就可以得出正确的角度值。`atan2()`和 `atan()`均返回弧度值。把弧度转换为度，只需将弧度值乘以 `180`，再除以 `pi` 即可。`pi` 的值通过计算表达式 `4*atan(1)`得到。程序清单 16.13 演示了这些步骤。另外，学习该程序还复习了结构和 `typedef` 相关的知识。

程序清单 16.14　rect_pol.c 程序

```
/* rect_pol.c -- 把直角坐标转换为极坐标 */
#include <stdio.h>
#include <math.h>

#define RAD_TO_DEG (180/(4 * atan(1)))

typedef struct polar_v {
    double magnitude;
    double angle;
} Polar_V;

typedef struct rect_v {
    double x;
    double y;
} Rect_V;

Polar_V rect_to_polar(Rect_V);

int main(void)
{
    Rect_V input;
    Polar_V result;

    puts("Enter x and y coordinates; enter q to quit:");
    while (scanf("%lf %lf", &input.x, &input.y) == 2)
    {
        result = rect_to_polar(input);
        printf("magnitude = %0.2f, angle = %0.2f\n",
            result.magnitude, result.angle);
    }
```

```
    puts("Bye.");

    return 0;
}

Polar_V rect_to_polar(Rect_V rv)
{
    Polar_V pv;

    pv.magnitude = sqrt(rv.x * rv.x + rv.y * rv.y);
    if (pv.magnitude == 0)
        pv.angle = 0.0;
    else
        pv.angle = RAD_TO_DEG * atan2(rv.y, rv.x);

    return pv;
}
```

下面是运行该程序后的一个输出示例：

```
Enter x and y coordinates; enter q to quit:
10 10
magnitude = 14.14, angle = 45.00
-12 -5
magnitude = 13.00, angle = -157.38
q
Bye.
```

如果编译时出现下面的消息：

```
Undefined: _sqrt
```

或

```
'sqrt': unresolved external
```

或者其他类似的消息，表明编译器链接器没有找到数学库。UNIX 系统会要求使用`-lm`标记（*flag*）指示链接器搜索数学库：

```
cc rect_pol.c –lm
```

注意，`-lm`标记在命令行的末尾。因为链接器在编译器编译 C 文件后才开始处理。在 Linux 中使用 GCC 编译器可能要这样写：

```
gcc rect_pol.c -lm
```

16.10.2　类型变体

基本的浮点型数学函数接受`double`类型的参数，并返回`double`类型的值。当然，也可以把`float`或`long double`类型的参数传递给这些函数，它们仍然能正常工作，因为这些类型的参数会被转换成`double`类型。这样做很方便，但并不是最好的处理方式。如果不需要双精度，那么用`float`类型的单精度值来计算会更快些。而且把`long double`类型的值传递给`double`类型的形参会损失精度，形参获得的值可能不是原来的值。为了解决这些潜在的问题，C 标准专门为`float`类型和`long double`类型提供了标准函数，即在原函数名前加上`f`或`l`前缀。因此，`sqrtf()`是`sqrt()`的`float`版本，`sqrtl()`是`sqrt()`的`long double`版本。

利用 C11 新增的泛型选择表达式定义一个泛型宏，根据参数类型选择最合适的数学函数版本。程序清单 16.15 演示了两种方法。

程序清单 16.15　generic.c 程序

```
//  generic.c  -- 定义泛型宏

#include <stdio.h>
#include <math.h>
#define RAD_TO_DEG (180/(4 * atanl(1)))

// 泛型平方根函数
#define SQRT(X) _Generic((X),\
    long double: sqrtl, \
    default: sqrt, \
    float: sqrtf)(X)

// 泛型正弦函数，角度的单位为度
#define SIN(X) _Generic((X),\
     long double: sinl((X)/RAD_TO_DEG),\
     default:    sin((X)/RAD_TO_DEG),\
     float:     sinf((X)/RAD_TO_DEG)\
)

int main(void)
{
     float x = 45.0f;
     double xx = 45.0;
     long double xxx = 45.0L;

     long double y = SQRT(x);
     long double yy = SQRT(xx);
     long double yyy = SQRT(xxx);
     printf("%.17Lf\n", y);     // 匹配 float
     printf("%.17Lf\n", yy);    // 匹配 default
     printf("%.17Lf\n", yyy);   // 匹配 long double
     int i = 45;
     yy = SQRT(i);              // 匹配 default
     printf("%.17Lf\n", yy);
     yyy = SIN(xxx);            // 匹配 long double
     printf("%.17Lf\n", yyy);

     return 0;
}
```

下面是该程序的输出：

```
6.70820379257202148
6.70820393249936942
6.70820393249936909
6.70820393249936942
0.70710678118654752
```

如上所示，SQRT(i)和 SQRT(xx)的返回值相同，因为它们的参数类型分别是 int 和 double，所以只能与 default 标签对应。

有趣的一点是，如何让_Generic 宏的行为像一个函数。SIN()的定义也许提供了一个方法：每个带标号的值都是函数调用，所以_Generic 表达式的值是一个特定的函数调用，如 sinf((X)/RAD_TO_DEG)，

用传入 SIN()的参数替换 X。

SQRT()的定义也许更简洁。_Generic 表达式的值就是函数名，如 sinf。函数的地址可以代替该函数名，所以_Generic 表达式的值是一个指向函数的指针。然而，紧随整个_Generic 表达式之后的是(X)，**函数指针(参数)**表示函数指针。因此，这是一个带指定的参数的函数指针。

简而言之，对于 SIN()，函数调用在泛型选择表达式内部；而对于 SQRT()，先对泛型选择表达式求值得一个指针，然后通过该指针调用它所指向的函数。

16.10.3　tgmath.h 库（C99）

C99 标准提供的 tgmath.h 头文件中定义了泛型类型宏，其效果与程序清单 16.15 类似。如果在 math.h 中为一个函数定义了 3 种类型（float、double 和 long double）的版本，那么 tgmath.h 文件就创建一个泛型类型宏，与原来 double 版本的函数名同名。例如，根据提供的参数类型，定义 sqrt()宏展开为 sqrtf()、sqrt()或 sqrtl()函数。换言之，sqrt()宏的行为和程序清单 16.15 中的 SQRT()宏类似。

如果编译器支持复数运算，就会支持 complex.h 头文件，其中声明了与复数运算相关的函数。例如，声明有 csqrtf()、csqrt()和 csqrtl()，这些函数分别返回 float complex、double complex 和 long double complex 类型的复数平方根。如果提供这些支持，那么 tgmath.h 中的 sqrt()宏也能展开为相应的复数平方根函数。

如果包含了 tgmath.h，要调用 sqrt()函数而不是 sqrt()宏，可以用圆括号把被调用的函数名括起来：

```
#include <tgmath.h>
...
    float x = 44.0;
    double y;
    y = sqrt(x);       // 调用宏，所以是 sqrtf(x)
    y = (sqrt)(x);     // 调用函数 sqrt()
```

这样做没问题，因为类函数宏的名称必须用圆括号括起来。圆括号只会影响操作顺序，不会影响括起来的表达式，所以这样做得到的仍然是函数调用的结果。实际上，在讨论函数指针时提到过，由于 C 语言奇怪而矛盾的函数指针规则，还也可以使用(*sqrt)()的形式来调用 sqrt()函数。

不借助 C 标准以外的机制，C11 新增的_Generic 表达式是实现 tgmath.h 最简单的方式。

16.11　通用工具库

通用工具库包含各种函数，包括随机数生成器、查找和排序函数、转换函数和内存管理函数。第 12 章介绍过 rand()、srand()、malloc()和 free()函数。在 ANSI C 标准中，这些函数的原型都在 stdlib.h 头文件中。附录 B 参考资料 V 列出了该系列的所有函数。现在，我们来进一步讨论其中的几个函数。

16.11.1　exit()和 atexit()函数

在前面的章节中我们已经在程序示例中用过 exit()函数。而且，在 main()返回系统时将自动调用 exit()函数。ANSI 标准还新增了一些不错的功能，其中最重要的是可以指定在执行 exit()时调用的特定函数。atexit()函数通过退出时注册被调用的函数提供这种功能，atexit()函数接受一个函数指针作为参数。程序清单 16.16 演示了它的用法。

程序清单 16.16　byebye.c 程序

```
/* byebye.c -- atexit()示例 */
#include <stdio.h>
#include <stdlib.h>
void sign_off(void);
void too_bad(void);

int main(void)
{
    int n;

    atexit(sign_off);      /* 注册 sign_off()函数 */
    puts("Enter an integer:");
    if (scanf("%d", &n) != 1)
    {
        puts("That's no integer!");
        atexit(too_bad);   /* 注册 too_bad()函数 */
        exit(EXIT_FAILURE);
    }
    printf("%d is %s.\n", n, (n % 2 == 0) ? "even" : "odd");

    return 0;
}

void sign_off(void)
{
    puts("Thus terminates another magnificent program from");
    puts("SeeSaw Software!");
}

void too_bad(void)
{
    puts("SeeSaw Software extends its heartfelt condolences");
    puts("to you upon the failure of your program.");
}
```

下面是该程序的一个运行示例：

```
Enter an integer:
212
212 is even.
Thus terminates another magnificent program from
SeeSaw Software!
```

如果在 IDE 中运行，可能看不到最后两行。下面是另一个运行示例：

```
Enter an integer:
what?
That's no integer!
SeeSaw Software extends its heartfelt condolences
to you upon the failure of your program.
Thus terminates another magnificent program from
SeeSaw Software!
```

在 IDE 中运行，可能看不到最后 4 行。

接下来，我们讨论 `atexit()`和 `exit()`的参数。

1. atexit()函数的用法

这个函数使用函数指针。要使用 atexit()函数，只需把退出时要调用的函数地址传递给 atexit()即可。函数名作为函数参数时相当于该函数的地址，所以该程序中把 sign_off 或 too_bad 作为参数。然后，atexit()注册函数列表中的函数，当调用 exit()时就会执行这些函数。ANSI 保证，在这个列表中至少可以放 32 个函数。最后调用 exit()函数时，exit()会执行这些函数（执行顺序与列表中的函数顺序相反，即最后添加的函数最先执行）。

注意，输入失败时，会调用 sign_off()和 too_bad()函数；但是输入成功时只会调用 sign_off()。因为只有输入失败时，才会进入 if 语句中注册 too_bad()。另外还要注意，最先调用的是最后一个被注册的函数。

atexit()注册的函数（如 sign_off()和 too_bad()）应该不带任何参数且返回类型为 void。通常，这些函数会执行一些清理任务，例如更新监视程序的文件或重置环境变量。

注意，即使没有显式调用 exit()，还是会调用 sign_off()，因为 main()结束时会隐式调用 exit()。

2. exit()函数的用法

exit()执行完 atexit()指定的函数后，会完成一些清理工作：刷新所有输出流、关闭所有打开的流和关闭由标准 I/O 函数 tmpfile()创建的临时文件。然后 exit()把控制权返回主机环境，如果可能的话，向主机环境报告终止状态。通常，UNIX 程序使用 0 表示成功终止，用非零值表示终止失败。UNIX 返回的代码并不适用于所有的系统，所以 ANSI C 为了可移植性的要求，定义了一个名为 EXIT_FAILURE 的宏表示终止失败。类似地，ANSI C 还定义了 EXIT_SUCCESS 表示成功终止。不过，exit()函数也接受 0 表示成功终止。在 ANSI C 中，在非递归的 main()中使用 exit()函数等价于使用关键字 return。尽管如此，在 main()以外的函数中使用 exit()也会终止整个程序。

16.11.2 qsort()函数

对较大型的数组而言，“快速排序”方法是最有效的排序算法之一。该算法由 C.A.R.Hoare 于 1962 年开发。它把数组不断分成更小的数组，直到变成单元素数组。首先，把数组分成两部分，一部分的值都小于另一部分的值。这个过程一直持续到数组完全排序好为止。

快速排序算法在 C 实现中的名称是 qsort()。qsort()函数排序数组的数据对象，其原型如下：

```
void qsort(void *base, size_t nmemb, size_t size,
           int (*compar)(const void *, const void *));
```

第 1 个参数是指针，指向待排序数组的首元素。ANSI C 允许把指向任何数据类型的指针强制转换成指向 void 的指针，因此，qsort()的第 1 个实际参数可以引用任何类型的数组。

第 2 个参数是待排序项的数量。函数原型把该值转换为 size_t 类型。前面提到过，size_t 定义在标准头文件中，是 sizeof 运算符返回的整数类型。

由于 qsort()把第 1 个参数转换为 void 指针，所以 qsort()不知道数组中每个元素的大小。为此，函数原型用第 3 个参数补偿这一信息，显式指明待排序数组中每个元素的大小。例如，如果排序 double 类型的数组，那么第 3 个参数应该是 sizeof(double)。

最后，qsort()还需要一个指向函数的指针，这个被指针指向的比较函数用于确定排序的顺序。该函数应接受两个参数：分别指向待比较两项的指针。如果第 1 项的值大于第 2 项，比较函数则返回正数；如果两项相同，则返回 0；如果第 1 项的值小于第 2 项，则返回负数。qsort()根据给定的其他信息计算出两个指针的值，然后把它们传递给比较函数。

qsort()原型中的第 4 个函数确定了比较函数的形式：

```
int (*compar)(const void *, const void *)
```

这表明 qsort()最后一个参数是一个指向函数的指针，该函数返回 int 类型的值且接受两个指向 const void 的指针作为参数，这两个指针指向待比较项。

程序清单 16.17 和后面的讨论解释了如何定义一个比较函数，以及如何使用 qsort()。该程序创建了一个内含随机浮点值的数组，并排序了这个数组。

程序清单 16.17　qsorter.c 程序

```
/* qsorter.c -- 用 qsort()排序一组数字 */
#include <stdio.h>
#include <stdlib.h>

#define NUM 40
void fillarray(double ar [], int n);
void showarray(const double ar [], int n);
int mycomp(const void * p1, const void * p2);

int main(void)
{
    double vals[NUM];
    fillarray(vals, NUM);
    puts("Random list:");
    showarray(vals, NUM);
    qsort(vals, NUM, sizeof(double), mycomp);
    puts("\nSorted list:");
    showarray(vals, NUM);
    return 0;
}

void fillarray(double ar [], int n)
{
    int index;

    for (index = 0; index < n; index++)
        ar[index] = (double) rand() / ((double) rand() + 0.1);
}

void showarray(const double ar [], int n)
{
    int index;

    for (index = 0; index < n; index++)
    {
        printf("%9.4f ", ar[index]);
        if (index % 6 == 5)
            putchar('\n');
    }
    if (index % 6 != 0)
        putchar('\n');
}

/* 按从小到大的顺序排序 */
int mycomp(const void * p1, const void * p2)
```

```
{
    /* 要使用指向 double 的指针来访问这两个值 */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;

    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

下面是该程序的运行示例：

```
Random list:
0.0001   1.6475   2.4332   0.0693   0.7268   0.7383
24.0357  0.1009   87.1828  5.7361   0.6079   0.6330
1.6058   0.1406   0.5933   1.1943   5.5295   2.2426
0.8364   2.7127   0.2514   0.9593   8.9635   0.7139
0.6249   1.6044   0.8649   2.1577   0.5420   15.0123
1.7931   1.6183   1.9973   2.9333   12.8512  1.3034
0.3032   1.1406   18.7880  0.9887

Sorted list:
0.0001   0.0693   0.1009   0.1406   0.2514   0.3032
0.5420   0.5933   0.6079   0.6249   0.6330   0.7139
0.7268   0.7383   0.8364   0.8649   0.9593    0.9887
1.1406   1.1943   1.3034   1.6044   1.6058   1.6183
1.6475   1.7931   1.9973   2.1577   2.2426   2.4332
2.7127   2.9333   5.5295   5.7361   8.9635   12.8512
15.0123  18.7880  24.0357  87.1828
```

接下来分析两点：qsort()的用法和 mycomp()的定义。

1. qsort()的用法

qsort()函数排序数组的数据对象。该函数的 ANSI 原型如下：

```
void qsort (void *base, size_t nmemb, size_t size,
            int (*compar)(const void *, const void *));
```

第 1 个参数值指向待排序数组首元素的指针。在该程序中，实际参数是 double 类型的数组名 vals，因此指针指向该数组的首元素。根据该函数的原型，参数 vals 会被强制转换成指向 void 的指针。由于 ANSI C 允许把指向任何数据类型的指针强制转换成指向 void 的指针，所以 qsort()的第 1 个实际参数可以引用任何类型的数组。

第 2 个参数是待排序项的数量。在程序清单 16.17 中是 NUM，即数组元素的数量。函数原型把该值转换为 size_t 类型。

第 3 个参数是数组中每个元素占用的空间大小，本例中为 sizeof(double)。

最后一个参数是 mycomp，这里函数名即是函数的地址，该函数用于比较元素。

2. mycomp()的定义

前面提到过，qsort()的原型中规定了比较函数的形式：

```
int (*compar)(const void *, const void *)
```

这表明 qsort()最后一个参数是一个指向函数的指针，该函数返回 int 类型的值且接受两个指向 const void 的指针作为参数。程序中 mycomp()使用的就是这个原型：

```
int mycomp(const void * p1, const void * p2);
```

记住，函数名作为参数时即是指向该函数的指针。因此，mycomp 与 compar 原型相匹配。

qsort()函数把两个待比较元素的地址传递给比较函数。在该程序中，把待比较的两个 double 类型值的地址赋给 p1 和 p2。注意，qsort()的第 1 个参数引用整个数组，比较函数中的两个参数引用数组中的两个元素。这里存在一个问题。为了比较指针所指向的值，必须解引用指针。因为值是 double 类型，所以要把指针解引用为 double 类型的值。然而，qsort()要求指针指向 void。要解决这个问题，必须在比较函数的内部声明两个类型正确的指针，并初始化它们分别指向作为参数传入的值：

```
/* 按从小到大的顺序排序值 */
int mycomp(const void * p1, const void * p2)
{
    /* 使用指向double类型的指针访问值 */
    const double * a1 = (const double *) p1;
    const double * a2 = (const double *) p2;
    if (*a1 < *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

简而言之，为了让该方法具有通用性，qsort()和比较函数使用了指向 void 的指针。因此，必须把数组中每个元素的大小明确告诉 qsort()，并且在比较函数的定义中，必须把该函数的指针参数转换为对具体应用而言类型正确的指针。

注意 C 和 C++中的 void*

C 和 C++对待指向 void 的指针有所不同。在这两种语言中，都可以把任何类型的指针赋给 void 类型的指针。例如，程序清单 16.17 中，qsort()的函数调用中把 double*指针赋给 void*指针。但是，C++要求在把 void*指针赋给任何类型的指针时必须进行强制类型转换。而 C 没有这样的要求。例如，程序清单 16.17 中的 mycomp()函数，就使用了这样的强制类型转换:

```
const double * a1 = (const double *) p1;
```

这种强制类型转换，在 C 中是可选的，但在 C++中是必须的。因为两种语言都使用强制类型转换，所以遵循 C++的要求也无不妥。将来如果要把该程序转成 C++，就不必更改这部分的代码。

下面再来看一个比较函数的例子。假设有下面的声明：

```
struct names {
    char first[40];
    char last[40];
};
struct names staff[100];
```

如何调用 qsort()？模仿程序清单 16.17 中 qsort()的函数调用，应该是这样：

```
qsort(staff, 100, sizeof(struct names), comp);
```

这里 comp 是比较函数的函数名。那么，应如何编写这个函数？假设要先按姓排序，如果同姓再按名

排序，可以这样编写该函数：

```
#include <string.h>
int comp(const void * p1, const void * p2) /* 该函数的形式必须是这样 */
{
    /* 得到正确类型的指针 */
    const struct names *ps1 = (const struct names *) p1;
    const struct names *ps2 = (const struct names *) p2;
    int res;
    res = strcmp(ps1->last, ps2->last); /* 比较姓 */
    if (res != 0)
        return res;
    else /* 如果同姓，则比较名 */
        return strcmp(ps1->first, ps2->first);
}
```

该函数使用 `strcmp()`函数进行比较。`strcmp()`的返回值与比较函数的要求相匹配。注意，通过指针访问结构成员时必须使用`->`运算符。

16.12 断言库

`assert.h` 头文件支持的断言库是一个用于辅助调试程序的小型库。它由 `assert()`宏组成，接受一个整型表达式作为参数。如果表达式求值为假（非零），`assert()`宏就在标准错误流（`stderr`）中写入一条错误信息，并调用 `abort()`函数终止程序（`abort()`函数的原型在 `stdlib.h` 头文件中）。`assert()`宏是为了标识出程序中某些条件为真的关键位置，如果其中的一个具体条件为假，就用 `assert()`语句终止程序。通常，`assert()`的参数是一个条件表达式或逻辑表达式。如果 `assert()`中止了程序，它首先会显示失败的测试、包含测试的文件名和行号。

16.12.1 assert 的用法

程序清单 16.18 演示了一个使用 `assert` 的小程序。在求平方根之前，该程序断言 `z` 是否大于或等于 `0`。程序还错误地减去一个值而不是加上一个值，故意让 `z` 得到不合适的值。

程序清单 16.18 assert.c 程序

```
/* assert.c -- 使用 assert() */
#include <stdio.h>
#include <math.h>
#include <assert.h>
int main()
{
    double x, y, z;

    puts("Enter a pair of numbers (0 0 to quit): ");
    while (scanf("%lf%lf", &x, &y) == 2
        && (x != 0 || y != 0))
    {
        z = x * x - y * y;  /* 应该用 + */
        assert(z >= 0);
        printf("answer is %f\n", sqrt(z));
        puts("Next pair of numbers: ");
    }
    puts("Done");
```

```
    return 0;
}
```

下面是该程序的运行示例：

```
Enter a pair of numbers (0 0 to quit):
4 3
answer is 2.645751
Next pair of numbers:
5 3
answer is 4.000000
Next pair of numbers:
3 5
Assertion failed: (z >= 0), function main, file /Users/assert.c, line 14.
```

具体的错误提示因编译器而异。让人困惑的是，这条消息可能不是指明 `z >= 0`，而是指明没有满足 `z >= 0` 的条件。

用 `if` 语句也能完成类似的任务：

```
if (z < 0)
{
    puts("z less than 0");
    abort();
}
```

但是，使用 `assert()`有几个好处：它不仅能自动标识文件和出问题的行号，还有一种无需更改代码就能开启或关闭 `assert()`的机制。如果认为已经排除了程序的 bug，就可以把下面的宏定义写在包含 `assert.h` 的位置前面：

```
#define NDEBUG
```

并重新编译程序，这样编译器就会禁用文件中的所有 `assert()`语句。如果程序又出现问题，可以移除这条`#define` 指令（或者把它注释掉），然后重新编译程序，这样就重新启用了 `assert()`语句。

16.12.2 `_Static_assert`（C11）

`assert()`表达式是在运行时进行检查。C11 新增了一个特性：`_Static_assert` 声明，可以在编译时检查 `assert()`表达式。因此，`assert()`可以导致正在运行的程序中止，而`_Static_assert()`可以导致程序无法通过编译。`_Static_assert()`接受两个参数。第 1 个参数是整型常量表达式，第 2 个参数是一个字符串。如果第 1 个表达式求值为 `0`（或`_False`），编译器会显示字符串，而且不编译该程序。看看程序清单 16.19 的小程序，然后查看 `assert()`和`_Static_assert()`的区别。

程序清单 16.19　`statasrt.c` 程序

```
//  statasrt.c
#include <stdio.h>
#include <limits.h>
_Static_assert(CHAR_BIT == 16, "16-bit char falsely assumed");
int main(void)
{
    puts("char is 16 bits.");
    return 0;
}
```

下面是在命令行编译的示例：

```
$ clang statasrt.c
statasrt.c:4:1: error: static_assert failed "16-bit char falsely assumed"
_Static_assert(CHAR_BIT == 16, "16-bit char falsely assumed");
^              ~~~~~~~~~~~~~~~
1 error generated.
$
```

根据语法，`_Static_assert()`被视为声明。因此，它可以出现在函数中，或者在这种情况下出现在函数的外部。

`_Static_assert` 要求它的第 1 个参数是整型常量表达式，这保证了能在编译期求值（`sizeof` 表达式被视为整型常量）。不能用程序清单 16.18 中的 `assert` 代替`_Static_assert`，因为 `assert` 中作为测试表达式的 `z > 0` 不是常量表达式，要到程序运行时才求值。当然，可以在程序清单 16.19 的 `main()`函数中使用 `assert(CHAR_BIT == 16)`，但这会在编译和运行程序后才生成一条错误信息，很没效率。

16.13　string.h 库中的 memcpy()和 memmove()

不能把一个数组赋给另一个数组，所以要通过循环把数组中的每个元素赋给另一个数组相应的元素。有一个例外的情况是：使用 `strcpy()`和 `strncpy()`函数来处理字符数组。`memcpy()`和 `memmove()`函数提供类似的方法处理任意类型的数组。下面是这两个函数的原型：

```
void *memcpy(void * restrict s1, const void * restrict s2, size_t n);
void *memmove(void *s1, const void *s2, size_t n);
```

这两个函数都从 `s2` 指向的位置拷贝 `n` 字节到 `s1` 指向的位置，而且都返回 `s1` 的值。所不同的是，`memcpy()`的参数带关键字 `restrict`，即 `memcpy()`假设两个内存区域之间没有重叠；而 `memmove()`不作这样的假设，所以拷贝过程类似于先把所有字节拷贝到一个临时缓冲区，然后再拷贝到最终目的地。如果使用 `memcpy()`时，两区域出现重叠会怎样？其行为是未定义的，这意味着该函数可能正常工作，也可能失败。编译器不会在本不该使用 `memcpy()`时禁止你使用，作为程序员，在使用该函数时有责任确保两个区域不重叠。

由于这两个函数设计用于处理任何数据类型，所有它们的参数都是两个指向 `void` 的指针。C 允许把任何类型的指针赋给 `void *`类型的指针。如此宽容导致函数无法知道待拷贝数据的类型。因此，这两个函数使用第 3 个参数指明待拷贝的字节数。注意，对数组而言，字节数一般与元素个数不同。如果要拷贝数组中 10 个 `double` 类型的元素，要使用 `10*sizeof(double)`，而不是 `10`。

程序清单 16.20 中的程序使用了这两个函数。该程序假设 `double` 类型是 `int` 类型的两倍大小。另外，该程序还使用了 `C11` 的`_Static_assert` 特性测试断言。

程序清单 16.20　mems.c 程序

```
// mems.c -- 使用 memcpy() 和 memmove()
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#define SIZE 10
void show_array(const int ar [], int n);
// 如果编译器不支持 C11 的_Static_assert，可以注释掉下面这行
_Static_assert(sizeof(double) == 2 * sizeof(int), "double not twice int size");
int main()
{
    int values[SIZE] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    int target[SIZE];
```

```
    double curious[SIZE / 2] = { 2.0, 2.0e5, 2.0e10, 2.0e20, 5.0e30 };

    puts("memcpy() used:");
    puts("values (original data): ");
    show_array(values, SIZE);
    memcpy(target, values, SIZE * sizeof(int));
    puts("target (copy of values):");
     show_array(target, SIZE);

    puts("\nUsing memmove() with overlapping ranges:");
    memmove(values + 2, values, 5 * sizeof(int));
    puts("values -- elements 0-4 copied to 2-6:");
    show_array(values, SIZE);

    puts("\nUsing memcpy() to copy double to int:");
    memcpy(target, curious, (SIZE / 2) * sizeof(double));
    puts("target -- 5 doubles into 10 int positions:");
    show_array(target, SIZE / 2);
    show_array(target + 5, SIZE / 2);

    return 0;
}

void show_array(const int ar [], int n)
{
    int i;

    for (i = 0; i < n; i++)
        printf("%d ", ar[i]);
    putchar('\n');
}
```

下面是该程序的输出：

```
memcpy() used:
values (original data):
1 2 3 4 5 6 7 8 9 10
target (copy of values):
1 2 3 4 5 6 7 8 9 10

Using memmove() with overlapping ranges:
values -- elements 0-4 copied to 2-6:
1 2 1 2 3 4 5 8 9 10

Using memcpy() to copy double to int:
target -- 5 doubles into 10 int positions:
0 1073741824 0 1091070464 536870912
1108516959 2025163840 1143320349 -2012696540 1179618799
```

程序中最后一次调用 memcpy()从 double 类型数组中把数据拷贝到 int 类型数组中，这演示了 memcpy()函数不知道也不关心数据的类型，它只负责从一个位置把一些字节拷贝到另一个位置（例如，从结构中拷贝数据到字符数组中）。而且，拷贝过程中也不会进行数据转换。如果用循环对数组中的每个元素赋值，double 类型的值会在赋值过程被转换为 int 类型的值。这种情况下，按原样拷贝字节，然后程序把这些位组合解释成 int 类型。

16.14　可变参数：stdarg.h

本章前面提到过变参宏，即该宏可以接受可变数量的参数。stdarg.h 头文件为函数提供了一个类似的功能，但是用法比较复杂。必须按如下步骤进行：

1．提供一个使用省略号的函数原型；

2．在函数定义中创建一个 va_list 类型的变量；

3．用宏把该变量初始化为一个参数列表；

4．用宏访问参数列表；

5．用宏完成清理工作。

接下来详细分析这些步骤。这种函数的原型应该有一个形参列表，其中至少有一个形参和一个省略号：

```
void f1(int n, ...);                // 有效
int f2(const char * s, int k, ...); // 有效
char f3(char c1, ..., char c2);// 无效，省略号不在最后
double f3(...);                     // 无效，没有形参
```

最右边的形参（即省略号的前一个形参）起着特殊的作用，标准中用 *parmN* 这个术语来描述该形参。在上面的例子中，第 1 行 f1()中 *parmN* 为 n，第 2 行 f2()中 *parmN* 为 k。传递给该形参的实际参数是省略号部分代表的参数数量。例如，可以这样使用前面声明的 f1()函数：

```
f1(2, 200, 400);           // 2 个额外的参数
f1(4, 13, 117, 18, 23);    // 4 个额外的参数
```

接下来，声明在 stdarg.h 中的 va_list 类型代表一种用于储存形参对应的形参列表中省略号部分的数据对象。变参函数的定义起始部分类似下面这样：

```
double sum(int lim,...)
{
     va_list ap;   //声明一个储存参数的对象
```

在该例中，lim 是 *parmN* 形参，它表明变参列表中参数的数量。

然后，该函数将使用定义在 stdarg.h 中的 va_start()宏，把参数列表拷贝到 va_list 类型的变量中。该宏有两个参数：va_list 类型的变量和 parmN 形参。接着上面的例子讨论，va_list 类型的变量是 ap，parmN 形参是 lim。所以，应这样调用它：

```
va_start(ap, lim);// 把 ap 初始化为参数列表
```

下一步是访问参数列表的内容，这涉及使用另一个宏 va_arg()。该宏接受两个参数：一个 va_list 类型的变量和一个类型名。第 1 次调用 va_arg()时，它返回参数列表的第 1 项；第 2 次调用时返回第 2 项，以此类推。表示类型的参数指定了返回值的类型。例如，如果参数列表中的第 1 个参数是 double 类型，第 2 个参数是 int 类型，可以这样做：

```
double tic;
int toc;
...
tic = va_arg(ap, double);  // 检索第 1 个参数
toc = va_arg(ap, int);     //检索第 2 个参数
```

注意，传入的参数类型必须与宏参数的类型相匹配。如果第 1 个参数是 10.0，上面 tic 那行代码可以正常工作。但是如果参数是 10，这行代码可能会出错。这里不会像赋值那样把 double 类型自动转换成 int 类型。

最后，要使用 va_end()宏完成清理工作。例如，释放动态分配用于储存参数的内存。该宏接受一个 va_list 类型的变量：

```
va_end(ap);   // 清理工作
```

调用 va_end(ap)后，只有用 va_start 重新初始化 ap 后，才能使用变量 ap。

因为 va_arg()不提供退回之前参数的方法，所以有必要保存 va_list 类型变量的副本。C99 新增了一个宏用于处理这种情况：va_copy()。该宏接受两个 va_list 类型的变量作为参数，它把第 2 个参数拷贝给第 1 个参数：

```
va_list ap;
va_list apcopy;
double
double tic;
int toc;
...
va_start(ap, lim);          // 把 ap 初始化为一个参数列表
va_copy(apcopy, ap);        // 把 apcopy 作为 ap 的副本
tic = va_arg(ap, double);   // 检索第 1 个参数
toc = va_arg(ap, int);      // 检索第 2 个参数
```

此时，即使删除了 ap，也可以从 apcopy 中检索两个参数。

程序清单 16.21 中的程序示例中演示了如何创建这样的函数，该函数对可变参数求和。sum()的第 1 个参数是待求和项的数目。

程序清单 16.21　varargs.c 程序

```
//varargs.c -- use variable number of arguments
#include <stdio.h>
#include <stdarg.h>
double sum(int, ...);

int main(void)
{
    double s, t;

    s = sum(3, 1.1, 2.5, 13.3);
    t = sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1);
    printf("return value for "
        "sum(3, 1.1, 2.5, 13.3):             %g\n", s);
    printf("return value for "
        "sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1): %g\n", t);

    return 0;
}

double sum(int lim, ...)
{
    va_list ap;                     // 声明一个对象储存参数
    double tot = 0;
    int i;

    va_start(ap, lim);          // 把 ap 初始化为参数列表
    for (i = 0; i < lim; i++)
        tot += va_arg(ap, double); // 访问参数列表中的每一项
```

```
    va_end(ap);                          // 清理工作

    return tot;
}
```

下面是该程序的输出：

```
return value for sum(3, 1.1, 2.5, 13.3):                   16.9
return value for sum(6, 1.1, 2.1, 13.1, 4.1, 5.1, 6.1):   31.6
```

查看程序中的运算可以发现，第 1 次调用 sum()时对 3 个数求和，第 2 次调用时对 6 个数求和。

总而言之，使用变参函数比使用变参宏更复杂，但是函数的应用范围更广。

16.15　关键概念

C 标准不仅描述 C 语言，还描述了组成 C 语言的软件包、C 预处理器和 C 标准库。通过预处理器可以控制编译过程、列出要替换的内容、指明要编译的代码行和影响编译器其他方面的行为。C 库扩展了 C 语言的作用范围，为许多编程问题提供现成的解决方案。

16.16　本章小结

C 预处理器和 C 库是 C 语言的两个重要的附件。C 预处理器遵循预处理器指令，在编译源代码之前调整源代码。C 库提供许多有助于完成各种任务的函数，包括输入、输出、文件处理、内存管理、排序与搜索、数学运算、字符串处理等。附录 B 的参考资料 V 中列出了完整的 ANSI C 库。

16.17　复习题

1. 下面的几组代码由一个或多个宏组成，其后是使用宏的源代码。在每种情况下代码的结果是什么？这些代码是否是有效代码？（假设其中的变量已声明）

   a.

   ```
   #define FPM 5280 /*每英里的英尺数*/
   dist = FPM * miles;
   ```

   b.

   ```
   #define FEET 4
   #define POD FEET + FEET
   plort = FEET * POD;
   ```

   c.

   ```
   #define SIX = 6;
   nex = SIX;
   ```

   d.

   ```
   #define NEW(X) X + 5
   y = NEW(y);
   berg = NEW(berg) * lob;
   est = NEW(berg) / NEW(y);
   nilp = lob * NEW(-berg);
   ```

2. 修改复习题 1 中 d 部分的定义，使其更可靠。

3. 定义一个宏函数，返回两值中的较小值。

4. 定义 EVEN_GT(X, Y)宏，如果 X 为偶数且大于 Y，该宏返回 1。
5. 定义一个宏函数，打印两个表达式及其值。例如，若参数为 3+4 和 4*12，则打印：

   3+4 is 7 and 4*12 is 48
6. 创建#define 指令完成下面的任务。

   a. 创建一个值为 25 的命名常量。

   b. SPACE 表示空格字符。

   c. PS()代表打印空格字符。

   d. BIG(X)代表 X 的值加 3。

   e. SUMSQ(X, Y)代表 X 和 Y 的平方和。
7. 定义一个宏，以下面的格式打印名称、值和 int 类型变量的地址：

   name: fop; value: 23; address: ff464016
8. 假设在测试程序时要暂时跳过一块代码，如何在不移除这块代码的前提下完成这项任务？
9. 编写一段代码，如果定义了 PR_DATE 宏，则打印预处理的日期。
10. 内联函数部分讨论了 3 种不同版本的 square()函数。从行为方面看，这 3 种版本的函数有何不同？
11. 创建一个使用泛型选择表达式的宏，如果宏参数是_Bool 类型，对"boolean"求值，否则对"not boolean"求值。
12. 下面的程序有什么错误？

```
#include <stdio.h>
int main(int argc, char argv[])
{
    printf("The square root of %f is %f\n", argv[1],sqrt(argv[1]) );
}
```

13. 假设 scores 是内含 1000 个 int 类型元素的数组，要按降序排序该数组中的值。假设你使用 qsort()和 comp()比较函数。

    a. 如何正确调用 qsort()？

    b. 如何正确定义 comp()？
14. 假设 data1 是内含 100 个 double 类型元素的数组，data2 是内含 300 个 double 类型元素的数组。

    a. 编写 memcpy()的函数调用，把 data2 中的前 100 个元素拷贝到 data1 中。

    b. 编写 memcpy()的函数调用，把 data2 中的后 100 个元素拷贝到 data1 中。

16.18 编程练习

1. 开发一个包含你需要的预处理器定义的头文件。
2. 两数的调和平均数这样计算：先得到两数的倒数，然后计算两个倒数的平均值，最后取计算结果的倒数。使用#define 指令定义一个宏“函数”，执行该运算。编写一个简单的程序测试该宏。
3. 极坐标用向量的模（即向量的长度）和向量相对 x 轴逆时针旋转的角度来描述该向量。直角坐标用向量的 x 轴和 y 轴的坐标来描述该向量（见图 16.3）。编写一个程序，读取向量的模和角度（单位：

度），然后显示 x 轴和 y 轴的坐标。相关方程如下：

```
x = r*cos A y = r*sin A
```

需要一个函数来完成转换，该函数接受一个包含极坐标的结构，并返回一个包含直角坐标的结构（或返回指向该结构的指针）。

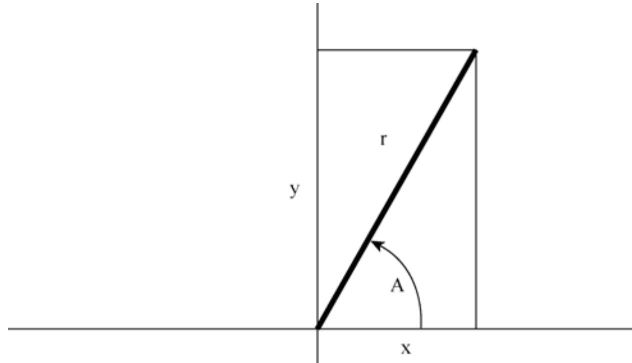


图 16.3　直角坐标和极坐标

4. ANSI 库这样描述 `clock()`函数的特性：

   ```
   #include <time.h>
   clock_t clock (void);
   ```

   这里，`clock_t` 是定义在 `time.h` 中的类型。该函数返回处理器时间，其单位取决于实现（如果处理器时间不可用或无法表示，该函数将返回`-1`）。然而，`CLOCKS_PER_SEC`（也定义在 `time.h` 中）是每秒处理器时间单位的数量。因此，两个 `clock()`返回值的差值除以 `CLOCKS_PER_SEC` 得到两次调用之间经过的秒数。在进行除法运算之前，把值的类型强制转换成 `double` 类型，可以将时间精确到小数点以后。编写一个函数，接受一个 `double` 类型的参数表示时间延迟数，然后在这段时间运行一个循环。编写一个简单的程序测试该函数。

5. 编写一个函数接受这些参数：内含 `int` 类型元素的数组名、数组的大小和一个代表选取次数的值。该函数从数组中随机选择指定数量的元素，并打印它们。每个元素只能选择一次（模拟抽奖数字或挑选陪审团成员）。另外，如果你的实现有 `time()`（第 12 章讨论过）或类似的函数，可在 `srand()`中使用这个函数的输出来初始化随机数生成器 `rand()`。编写一个简单的程序测试该函数。

6. 修改程序清单 16.17，使用 `struct names` 元素（在程序清单 16.17 后面的讨论中定义过），而不是 `double` 类型的数组。使用较少的元素，并用选定的名字显式初始化数组。

7. 下面是使用变参函数的一个程序段：

   ```
   #include <stdio.h>
   #include <stdlib.h>
   #include <stdarg.h>
   void show_array(const double ar[], int n);
   double * new_d_array(int n, ...);

   int main()
   {
       double * p1;
       double * p2;

       p1 = new_d_array(5, 1.2, 2.3, 3.4, 4.5, 5.6);
       p2 = new_d_array(4, 100.0, 20.00, 8.08, -1890.0);
   ```

```
    show_array(p1, 5);
    show_array(p2, 4);
    free(p1);
    free(p2);

    return 0;
}
```

new_d_array()函数接受一个 int 类型的参数和 double 类型的参数。该函数返回一个指针，指向由 malloc()分配的内存块。int 类型的参数指定了动态数组中的元素个数，double 类型的值用于初始化元素（第 1 个值赋给第 1 个元素，以此类推）。编写 show_array()和 new_d_array()函数的代码，完成这个程序。

第 17 章

高级数据表示

本章介绍以下内容：

- 函数：进一步学习 malloc()
- 使用 C 表示不同类型的数据
- 新的算法，从概念上增强开发程序的能力
- 抽象数据类型（ADT）

学习计算机语言和学习音乐、木工或工程学一样。首先，要学会使用工具：学习如何演奏音阶、如何使用锤子等，然后解决各种问题，如降落、滑行以及平衡物体之类。到目前为止，读者一直在本书中学习和练习各种编程技能，如创建变量、结构、函数等。然而，如果想提高到更高层次时，工具是次要的，真正的挑战是设计和创建一个项目。本章将重点介绍这个更高的层次，教会读者如何把项目看作一个整体。本章涉及的内容可能比较难，但是这些内容非常有价值，将帮助读者从编程新手成长为老手。

我们先从程序设计的关键部分，即程序表示数据的方式开始。通常，程序开发最重要的部分是找到程序中表示数据的好方法。正确地表示数据可以更容易地编写程序其余部分。到目前为止，读者应该很熟悉 C 的内置类型：简单变量、数组、指针、结构和联合。

然而，找出正确的数据表示不仅仅是选择一种数据类型，还要考虑必须进行哪些操作。也就是说，必须确定如何储存数据，并且为数据类型定义有效的操作。例如，C 实现通常把 int 类型和指针类型都储存为整数，但是这两种类型的有效操作不相同。例如，两个整数可以相乘，但是两个指针不能相乘；可以用*运算符解引用指针，但是对整数这样做毫无意义。C 语言为它的基本类型都定义了有效的操作。但是，当你要设记数据表示的方案时，你可能需要自己定义有效操作。在 C 语言中，可以把所需的操作设计成 C 函数来表示。简而言之，设计一种数据类型包括设计如何储存该数据类型和设计一系列管理该数据的函数。

本章还会介绍一些算法（*algorithm*），即操控数据的方法。作为一名程序员，应该掌握这些可以反复解决类似问题的处理方法。

本章将进一步研究设计数据类型的过程，这是一个把算法和数据表示相匹配的过程。期间会用到一些常见的数据形式，如队列、列表和二叉树。

本章还将介绍抽象数据类型（ADT）的概念。抽象数据类型以面向问题而不是面向语言的方式，把解决问题的方法和数据表示结合起来。设计一个 ADT 后，可以在不同的环境中复用。理解 ADT 可以为将来学习面向对象程序设计（OOP）以及 C++语言做好准备。

17.1 研究数据表示

我们先从数据开始。假设要创建一个地址簿程序。应该使用什么数据形式储存信息？由于储存的每一项都包含多种信息，用结构来表示每一项很合适。如何表示多个项？是否用标准的结构数组？还是动态数组？还是一些其他形式？各项是否按字母顺序排列？是否要按照邮政编码（或地区编码）查找各项？需要

执行的行为将影响如何储存信息？简而言之，在开始编写代码之前，要在程序设计方面做很多决定。

如何表示储存在内存中的位图图像？位图图像中的每个像素在屏幕上都单独设置。在以前黑白屏的年代，可以使用一个计算机位（1 或 0）来表示一个像素点（开或闭），因此称之为位图。对于彩色显示器而言，如果 8 位表示一个像素，可以得到 256 种颜色。现在行业标准已发展到 65536 色（每像素 16 位）、16777216 色（每像素 24 位）、2147483 色（每像素 32 位），甚至更多。如果有 32 位色，且显示器有 2560×1440 的分辨率，则需要将近 1.18 亿位（14M）来表示一个屏幕的位图图像。是用这种方法表示，还是开发一种压缩信息的方法？是有损压缩（丢失相对次要的数据）还是无损压缩（没有丢失数据）？再次提醒读者注意，在开始编写代码之前，需要做很多程序设计方面的决定。

我们来处理一个数据表示的示例。假设要编写一个程序，让用户输入一年内看过的所有电影（包括 DVD 和蓝光光碟）。要储存每部影片的各种信息，如片名、发行年份、导演、主演、片长、影片的种类（喜剧、科幻、爱情等）、评级等。建议使用一个结构储存每部电影，一个数组储存一年内看过的电影。为简单起见，我们规定结构中只有两个成员：片名和评级（0～10）。程序清单 17.1 演示了一个基本的实现。

程序清单 17.1　films1.c 程序

```
/* films1.c -- 使用一个结构数组 */
#include <stdio.h>
#include <string.h>
#define TSIZE     45   /* 储存片名的数组大小 */
#define FMAX      5    /* 影片的最大数量 */

struct film {
    char title[TSIZE];
    int rating;
};
char * s_gets(char str[], int lim);
int main(void)
{
    struct film movies[FMAX];
    int i = 0;
    int j;

    puts("Enter first movie title:");
    while (i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&
        movies[i].title[0] != '\0')
    {
        puts("Enter your rating <0-10>:");
        scanf("%d", &movies[i++].rating);
        while (getchar() != '\n')
            continue;
        puts("Enter next movie title (empty line to stop):");
    }
    if (i == 0)
        printf("No data entered. ");
    else
        printf("Here is the movie list:\n");

    for (j = 0; j < i; j++)
        printf("Movie: %s  Rating: %d\n", movies[j].title,movies[j].rating);
    printf("Bye!\n");

    return 0;
```

```
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // 查找换行符
        if (find)                   // 如果地址不是 NULL,
            *find = '\0';           // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;           // 处理剩余输入行
    }
    return ret_val;
}
```

该程序创建了一个结构数组，然后把用户输入的数据储存在数组中。直到数组已满（用 FMAX 进行判断）或者到达文件结尾（用 NULL 进行判断），或者用户在首行按下 Enter 键（用'\0'进行判断），输入才会终止。

这样设计程序有点问题。首先，该程序很可能会浪费许多空间，因为大部分的片名都不会超过 40 个字符。但是，有些片名的确很长，如 *The Discreet Charm of the Bourgeoisie* 和 *Won Ton Ton, The Dog Who Saved Hollywood*。其次，许多人会觉得每年 5 部电影的限制太严格了。当然，也可以放宽这个限制，但是，要多大才合适？有些人每年可以看 500 部电影，因此可以把 FMAX 改为 500。但是，对有些人而言，这可能仍然不够，而对有些人而言一年根本看不了这么多部电影，这样就浪费了大量的内存。另外，一些编译器对自动存储类别变量（如 movies）可用的内存数量设置了一个默认的限制，如此大型的数组可能会超过默认设置的值。可以把数组声明为静态或外部数组，或者设置编译器使用更大的栈来解决这个问题。但是，这样做并不能根本解决问题。

该程序真正的问题是，数据表示太不灵活。程序在编译时确定所需内存量，其实在运行时确定会更好。要解决这个问题，应该使用动态内存分配来表示数据。可以这样做：

```
#define TSIZE 45   /*储存片名的数组大小*/
    struct film {
    char title[TSIZE];
    int rating;
};
...
int n, i;
struct film * movies;  /* 指向结构的指针 */
...
printf("Enter the maximum number of movies you'll enter:\n");
scanf("%d", &n);
movies = (struct film *) malloc(n * sizeof(struct film));
```

第 12 章介绍过，可以像使用数组名那样使用指针 movies。

```
while (i < FMAX && s_gets(movies[i].title, TSIZE) != NULL &&movies[i].title[0] != '\0')
```

使用 malloc()，可以推迟到程序运行时才确定数组中的元素数量。所以，如果只需要 20 个元素，程

序就不必分配存放 500 个元素的空间。但是，这样做的前提是，用户要为元素个数提供正确的值。

17.2　从数组到链表

理想的情况是，用户可以不确定地添加数据（或者不断添加数据直到用完内存量），而不是先指定要输入多少项，也不用让程序分配多余的空间。这可以通过在输入每一项后调用 malloc()分配正好能储存该项的空间。如果用户输入 3 部影片，程序就调用 malloc()3 次；如果用户输入 300 部影片，程序就调用 malloc()300 次。

不过，我们又制造了另一个麻烦。比较一下，一种方法是调用 malloc()一次，为 300 个 filem 结构请求分配足够的空间；另一种方法是调用 malloc()300 次，分别为每个 file 结构请求分配足够的空间。前者分配的是连续的内存块，只需要一个单独的指向 struct 变量（film）的指针，该指针指向已分配块中的第 1 个结构。简单的数组表示法让指针访问块中的每个结构，如前面代码段所示。第 2 种方法的问题是，无法保证每次调用 malloc()都能分配到连续的内存块。这意味着结构不一定被连续储存（见图 17.1）。因此，与第 1 种方法储存一个指向 300 个结构块的指针相比，你需要储存 300 个指针，每个指针指向一个单独储存的结构。

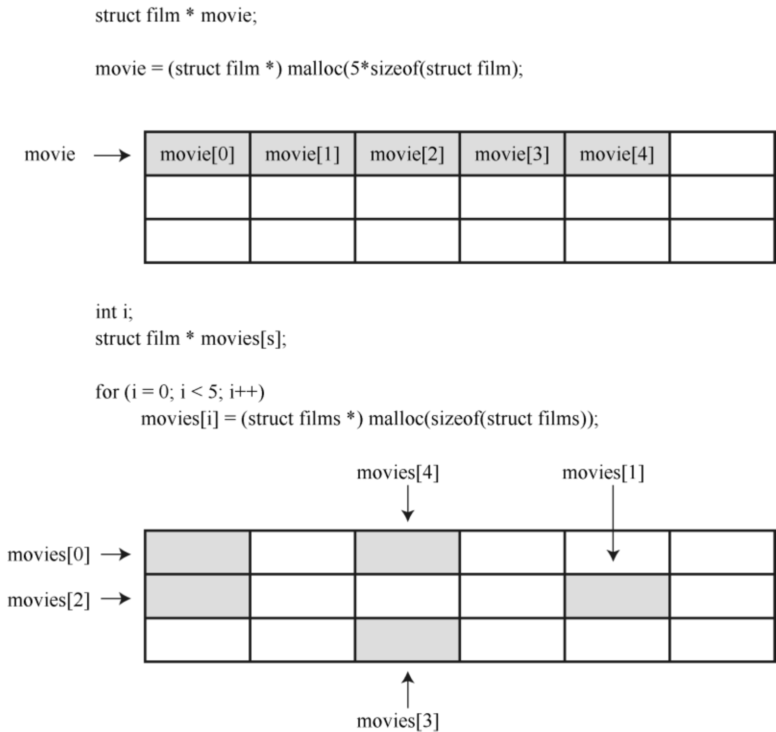


图 17.1　一块内存中分配结构和单独分配结构

一种解决方法是创建一个大型的指针数组，并在分配新结构时逐个给这些指针赋值，但是我们不打算使用这种方法：

```
#define TSIZE 45   /*储存片名的数组大小*/
#define FMAX 500   /*影片的最大数量*/
struct film {
    char title[TSIZE];
```

```
    int rating;
};
...
struct film * movies[FMAX]; /* 结构指针数组 */
int i;
...
movies[i] = (struct film *) malloc (sizeof (struct film));
```

如果用不完 500 个指针，这种方法节约了大量的内存，因为内含 500 个指针的数组比内含 500 个结构的数组所占的内存少得多。尽管如此，如果用不到 500 个指针，还是浪费了不少空间。而且，这样还是有 500 个结构的限制。

还有一种更好的方法。每次使用 `malloc()`为新结构分配空间时，也为新指针分配空间。但是，还得需要另一个指针来跟踪新分配的指针，用于跟踪新指针的指针本身，也需要一个指针来跟踪，以此类推。要重新定义结构才能解决这个潜在的问题，即每个结构中包含指向 `next` 结构的指针。然后，当创建新结构时，可以把该结构的地址储存在上一个结构中。简而言之，可以这样定义 `film` 结构：

```
#define TSIZE 45   /* 储存片名的数组大小*/
struct film {
    char title[TSIZE];
    int rating;
    struct film * next;
};
```

虽然结构不能含有与本身类型相同的结构，但是可以含有指向同类型结构的指针。这种定义是定义链表（*linked list*）的基础，链表中的每一项都包含着在何处能找到下一项的信息。

在学习链表的代码之前，我们先从概念上理解一个链表。假设用户输入的片名是 `Modern Times`，等级为 `10`。程序将为 `film` 类型的结构分配空间，把字符串 `Modern Times` 拷贝到结构中的 `title` 成员中，然后设置`rating`成员为`10`。为了表明该结构后面没有其他结构，程序要把`next`成员指针设置为`NULL`（`NULL`是一个定义在 `stdio.h` 头文件中的符号常量，表示空指针）。当然，还需要一个单独的指针储存第 1 个结构的地址，该指针被称为头指针（*head pointer*）。头指针指向链表中的第 1 项。图 17.2 演示了这种结构（为节约图片空间，压缩了 `title` 成员中的空白）。

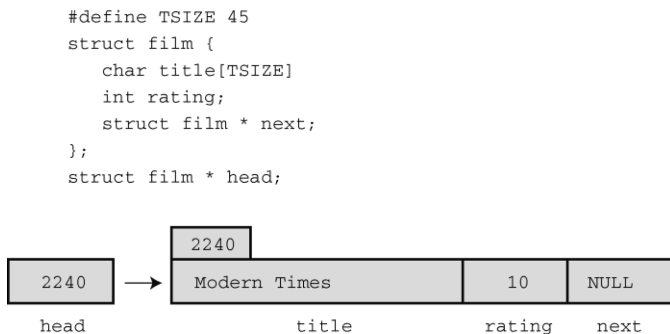


图 17.2 链表中的第 1 个项

现在，假设用户输入第 2 部电影及其评级，如 `Midnight in Paris` 和 `8`。程序为第 2 个 `film` 类型结构分配空间，把新结构的地址储存在第 1 个结构的 `next` 成员中（擦写了之前储存在该成员中的 `NULL`），这样链表中第 1 个结构中的 `next` 指针指向第 2 个结构。然后程序把 `Midnight in Paris` 和 `8` 拷贝到新结构中，并把第 2 个结构中的 `next` 成员设置为 `NULL`，表明该结构是链表中的最后一个结构。图 17.3 演示了这两个项。

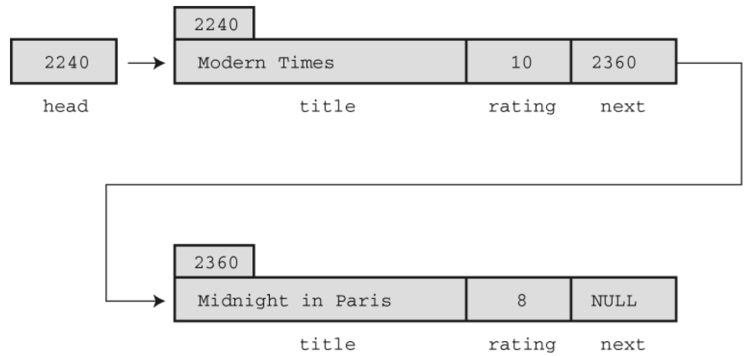


图 17.3　链表中的两个项

每加入一部新电影，就以相同的方式来处理。新结构的地址将储存在上一个结构中，新信息储存在新结构中，而且新结构中的 next 成员设置为 NULL。从而建立起如图 17.4 所示的链表。

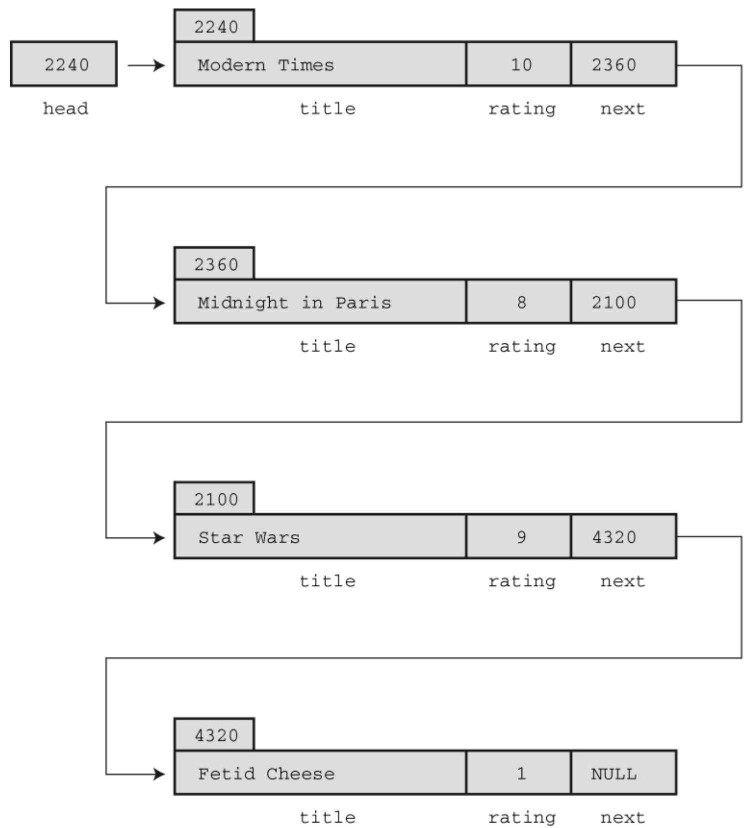


图 17.4　链表中的多个项

假设要显示这个链表，每显示一项，就可以根据该项中已储存的地址来定位下一个待显示的项。然而，这种方案能正常运行，还需要一个指针储存链表中第 1 项的地址，因为链表中没有其他项储存该项的地址。此时，头指针就派上了用场。

17.2.1　使用链表

从概念上了解了链表的工作原理，接着我们来实现它。程序清单 17.2 修改了程序清单 17.1，用链表而

不是数组来储存电影信息。

程序清单 17.2 `films2.c` **程序**

```
/* films2.c -- 使用结构链表 */
#include <stdio.h>
#include <stdlib.h>        /* 提供malloc()原型 */
#include <string.h>        /* 提供strcpy()原型 */
#define TSIZE   45         /* 储存片名的数组大小 */

struct film {
    char title[TSIZE];
    int rating;
    struct film * next;    /* 指向链表中的下一个结构 */
};
char * s_gets(char * st, int n);

int main(void)
{
    struct film * head = NULL;
    struct film * prev, *current;
    char input[TSIZE];

/* 收集并储存信息 */
    puts("Enter first movie title:");
    while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
    {
        current = (struct film *) malloc(sizeof(struct film));
        if (head == NULL)       /* 第 1 个结构 */
            head = current;
        else                    /* 后续的结构 */
            prev->next = current;
        current->next = NULL;
        strcpy(current->title, input);
        puts("Enter your rating <0-10>:");
        scanf("%d", &current->rating);
        while (getchar() != '\n')
            continue;
        puts("Enter next movie title (empty line to stop):");
        prev = current;
    }

    /* 显示电影列表 */
    if (head == NULL)
        printf("No data entered. ");
    else
        printf("Here is the movie list:\n");
    current = head;
    while (current != NULL)
    {
        printf("Movie: %s  Rating: %d\n",
            current->title, current->rating);
        current = current->next;
    }
```

```
    /* 完成任务，释放已分配的内存 */
    current = head;
    while (current != NULL)
    {
        current = head;
        head = current->next;
        free(current);
    }
    printf("Bye!\n");

    return 0;
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';           // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;      // 处理剩余输入行
    }
    return ret_val;
}
```

该程序用链表执行两个任务。第 1 个任务是，构造一个链表，把用户输入的数据储存在链表中。第 2 个任务是，显示链表。显示链表的任务比较简单，所以我们先来讨论它。

1. 显示链表

显示链表从设置一个指向第 1 个结构的指针（名为 current）开始。由于头指针（名为 head）已经指向链表中的第 1 个结构，所以可以用下面的代码来完成：

```
current = head;
```

然后，可以使用指针表示法访问结构的成员：

```
printf("Movie: %s Rating: %d\n", current->title, current->rating);
```

下一步是根据储存在该结构中 next 成员中的信息，重新设置 current 指针指向链表中的下一个结构。代码如下：

```
current = current->next;
```

完成这些之后，再重复整个过程。当显示到链表中最后一个项时，current 将被设置为 NULL，因为这是链表最后一个结构中 next 成员的值。

```
while (current != NULL)
{
    printf("Movie: %s Rating: %d\n", current->title, current->rating);
    current = current->next;
}
```

遍历链表时，为何不直接使用 head 指针，而要重新创建一个新指针（current）？因为如果使用 head 会改变 head 中的值，程序就找不到链表的开始处。

2. 创建链表

创建链表涉及下面 3 步：

（1）使用 malloc()为结构分配足够的空间；

（2）储存结构的地址；

（3）把当前信息拷贝到结构中。

如无必要不用创建一个结构，所以程序使用临时存储区（input 数组）获取用户输入的电影名。如果用户通过键盘模拟 EOF 或输入一行空行，将退出下面的循环：

```
while (s_gets(input, TSIZE) != NULL && input[0] != '\0')
```

如果用户进行输入，程序就分配一个结构的空间，并将其地址赋给指针变量 current：

```
current = (struct film *) malloc(sizeof(struct film));
```

链表中第 1 个结构的地址应储存在指针变量 head 中。随后每个结构的地址应储存在其前一个结构的 next 成员中。因此，程序要知道它处理的是否是第 1 个结构。最简单的方法是在程序开始时，把 head 指针初始化为 NULL。然后，程序可以使用 head 的值进行判断：

```
if (head == NULL) /* 第 1 个结构*/
    head = current;
else /* subsequent structures */
    prev->next = current;
```

在上面的代码中，指针 prev 指向上一次分配的结构。

接下来，必须为结构成员设置合适的值。尤其是，把 next 成员设置为 NULL，表明当前结构是链表的最后一个结构。还要把 input 数组中的电影名拷贝到 title 成员中，而且要给 rating 成员提供一个值。如下代码所示：

```
current->next = NULL;
strcpy(current->title, input);
puts("Enter your rating <0-10>:");
scanf("%d", &current->rating);
```

由于 s_gets()限制了只能输入 TSIZE-1 个字符，所以用 strcpy()函数把 input 数组中的字符串拷贝到 title 成员很安全。

最后，要为下一次输入做好准备。尤其是，要设置 prev 指向当前结构。因为在用户输入下一部电影且程序为新结构分配空间后，当前结构将成为新结构的上一个结构，所以程序在循环末尾这样设置该指针：

```
prev = current;
```

程序是否能正常运行？下面是该程序的一个运行示例：

```
Enter first movie title:
Spirited Away
Enter your rating <0-10>:
9
Enter next movie title (empty line to stop):
The Duelists
Enter your rating <0-10>:
8
Enter next movie title (empty line to stop):
Devil Dog: The Mound of Hound
Enter your rating <0-10>:
```

```
1
Enter next movie title (empty line to stop):

Here is the movie list:
Movie: Spirited Away Rating: 9
Movie: The Duelists Rating: 8
Movie: Devil Dog: The Mound of Hound Rating: 1
Bye!
```

3. 释放链表

在许多环境中，程序结束时都会自动释放 malloc()分配的内存。但是，最好还是成对调用 malloc()和 free()。因此，程序在清理内存时为每个已分配的结构都调用了 free()函数：

```
current = head;
while (current != NULL)
{
    current = head;
    head = current->next;
    free(current);
}
```

17.2.2　反思

films2.c 程序还有些不足。例如，程序没有检查 malloc()是否成功请求到内存，也无法删除链表中的项。这些不足可以弥补。例如，添加代码检查 malloc()的返回值是否是 NULL（返回 NULL 说明未获得所需内存）。如果程序要删除链表中的项，还要编写更多的代码。

这种用特定方法解决特定问题，并且在需要时才添加相关功能的编程方式通常不是最好的解决方案。另一方面，通常都无法预料程序要完成的所有任务。随着编程项目越来越大，一个程序员或编程团队事先计划好一切模式，越来越不现实。很多成功的大型程序都是由成功的小型程序逐步发展而来。

如果要修改程序，首先应该强调最初的设计，并简化其他细节。程序清单 17.2 中的程序示例没有遵循这个原则，它把概念模型和代码细节混在一起。例如，该程序的概念模型是在一个链表中添加项，但是程序却把一些细节（如，malloc()和 current->next 指针）放在最明显的位置，没有突出接口。如果程序能以某种方式强调给链表添加项，并隐藏具体的处理细节（如调用内存管理函数和设置指针）会更好。把用户接口和代码细节分开的程序，更容易理解和更新。学习下面的内容就可以实现这些目标。

17.3　抽象数据类型（ADT）

在编程时，应该根据编程问题匹配合适的数据类型。例如，用 int 类型代表你有多少双鞋，用 float 或 double 类型代表每双鞋的价格。在前面的电影示例中，数据构成了链表，每个链表项由电影名（C 字符串）和评级（一个 int 类型值）。C 中没有与之匹配的基本类型，所以我们定义了一个结构代表单独的项，然后设计了一些方法把一系列结构构成一个链表。本质上，我们使用 C 语言的功能设计了一种符合程序要求的新数据类型。但是，我们的做法并不系统。现在，我们用更系统的方法来定义数据类型。

什么是类型？类型特指两类信息：属性和操作。例如，int 类型的属性是它代表一个整数值，因此它共享整数的属性。允许对 int 类型进行算术操作是：改变 int 类型值的符号、两个 int 类型值相加、相减、相乘、相除、求模。当声明一个 int 类型的变量时，就表明了只能对该变量进行这些操作。

注意　整数属性

C 的 `int` 类型背后是一个更抽象的整数概念。数学家已经用正式的抽象方式定义了整数的属性。例如，假设 N 和 M 是整数，那么 N+M=M+N；假设 S、Q 也是整数，如果 N+M=S，而且 N+Q=S，那么 M=Q。可以认为数学家提供了整数的抽象概念，而 C 则实现了这一抽象概念。注意，实现整数的算术运算是表示整数必不可少的部分。如果只是储存值，并未在算术表达式中使用，`int` 类型就没那么有用了。还要注意的是，C 并未很好地实现整数。例如，整数是无穷大的数，但是 2 字节的 `int` 类型只能表示 65536 个整数。因此，不要混淆抽象概念和具体的实现。

假设要定义一个新的数据类型。首先，必须提供储存数据的方法，例如设计一个结构。其次，必须提供操控数据的方法。例如，考虑 `films2.c` 程序（程序清单 17.2）。该程序用链接的结构来储存信息，而且通过代码实现了如何添加和显示信息。尽管如此，该程序并未清楚地表明正在创建一个新类型。我们应该怎么做？

计算机科学领域已开发了一种定义新类型的好方法，用 3 个步骤完成从抽象到具体的过程。

1．提供类型属性和相关操作的抽象描述。这些描述既不能依赖特定的实现，也不能依赖特定的编程语言。这种正式的抽象描述被称为*抽象数据类型*（ADT）。

2．开发一个实现 ADT 的编程接口。也就是说，指明如何储存数据和执行所需操作的函数。例如在 C 中，可以提供结构定义和操控该结构的函数原型。这些作用于用户定义类型的函数相当于作用于 C 基本类型的内置运算符。需要使用该新类型的程序员可以使用这个接口进行编程。

3．编写代码实现接口。这一步至关重要，但是使用该新类型的程序员无需了解具体的实现细节。

我们再次以前面的电影项目为例来熟悉这个过程，并用新方法重新完成这个示例。

17.3.1　建立抽象

从根本上看，电影项目所需的是一个项链表。每一项包含电影名和评级。你所需的操作是把新项添加到链表的末尾和显示链表中的内容。我们把需要处理这些需求的抽象类型叫作*链表*。链表具有哪些属性？首先，链表应该能储存一系列的项。也就是说，链表能储存多个项，而且这些项以某种方式排列，这样才能描述链表的第 1 项、第 2 项或最后一项。其次，链表类型应该提供一些操作，如在链表中添加新项。下面是链表的一些有用的操作：

- 初始化一个空链表；
- 在链表末尾添加一个新项；
- 确定链表是否为空；
- 确定链表是否已满；
- 确定链表中的项数；
- 访问链表中的每一项执行某些操作，如显示该项。

对该电影项目而言，暂时不需要其他操作。但是一般的链表还应包含以下操作：

- 在链表的任意位置插入一个项；
- 移除链表中的一个项；
- 在链表中检索一个项（不改变链表）；

- 用另一个项替换链表中的一个项；
- 在链表中搜索一个项。

非正式但抽象的链表定义是：链表是一个能储存一系列项且可以对其进行所需操作的数据对象。该定义既未说明链表中可以储存什么项，也未指定是用数组、结构还是其他数据形式来储存项，而且并未规定用什么方法来实现操作（如，查找链表中元素的个数）。这些细节都留给实现完成。

为了让示例尽量简单，我们采用一种简化的链表作为抽象数据类型。它只包含电影项目中的所需属性。该类型总结如下：

类型名： 简单链表

类型属性： 可以储存一系列项

类型操作： 初始化链表为空
确定链表为空
确定链表已满
确定链表中的项数
在链表末尾添加项
遍历链表，处理链表中的项
清空链表

下一步是为开发简单链表 ADT 开发一个 C 接口。

17.3.2 建立接口

这个简单链表的接口有两个部分。第 1 部分是描述如何表示数据，第 2 部分是描述实现 ADT 操作的函数。例如，要设计在链表中添加项的函数和报告链表中项数的函数。接口设计应尽量与 ADT 的描述保持一致。因此，应该用某种通用的`Item`类型而不是一些特殊类型，如`int`或`struct film`。可以用C的`typedef`功能来定义所需的`Item`类型：

```
#define TSIZE 45   /* 储存电影名的数组大小 */
struct film
{
    char title[TSIZE];
    int rating;
};
typedef struct film Item;
```

然后，就可以在定义的其余部分使用 `Item` 类型。如果以后需要其他数据形式的链表，可以重新定义 `Item` 类型，不必更改其余的接口定义。

定义了 `Item` 之后，现在必须确定如何储存这种类型的项。实际上这一步属于实现步骤，但是现在决定好可以让示例更简单些。在 `films2.c` 程序中用链接的结构处理得很好，所以，我们在这里也采用相同的方法：

```
typedef struct node
{
    Item item;
    struct node * next;
} Node;
typedef Node * List;
```

在链表的实现中，每一个链节叫作节点（*node*）。每个节点包含形成链表内容的信息和指向下一个节点

的指针。为了强调这个术语，我们把 node 作为节点结构的标记名，并使用 typedef 把 Node 作为 struct node 结构的类型名。最后，为了管理链表，还需要一个指向链表开始处的指针，我们使用 typedef 把 List 作为该类型的指针名。因此，下面的声明：

```
List movies;
```

创建了该链表所需类型的指针 movies。

这是否是定义 List 类型的唯一方法？不是。例如，还可以添加一个变量记录项数：

```
typedef struct list
{
    Node * head;   /* 指向链表头的指针 */
    int size;      /* 链表中的项数 */
} List;            /* List 的另一种定义 */
```

可以像稍后的程序示例中那样，添加第 2 个指针储存链表的末尾。现在，我们还是使用 List 类型的第 1 种定义。这里要着重理解下面的声明创建了一个链表，而不一个指向节点的指针或一个结构：

```
List movies;
```

movies 代表的确切数据应该是接口层次不可见的实现细节。

例如，程序启动后应把头指针初始化为 NULL。但是，不要使用下面这样的代码：

```
movies = NULL;
```

为什么？因为稍后你会发现 List 类型的结构实现更好，所以应这样初始化：

```
movies.next = NULL;
movies.size = 0;
```

使用 List 的人都不用担心这些细节，只要能使用下面的代码就行：

```
InitializeList(movies);
```

使用该类型的程序员只需知道用 InitializeList()函数来初始化链表，不必了解 List 类型变量的实现细节。这是*数据隐藏*的一个示例，数据隐藏是一种从编程的更高层次隐藏数据表示细节的艺术。

为了指导用户使用，可以在函数原型前面提供以下注释：

```
/* 操作：初始化一个链表           */
/* 前提条件：plist 指向一个链表 */
/* 后置条件：该链表初始化为空       */
void InitializeList(List * plist);
```

这里要注意 3 点。第 1，注释中的“前提条件”（*precondition*）是调用该函数前应具备的条件。例如，需要一个待初始化的链表。第 2，注释中的“后置条件”（*postcondition*）是执行完该函数后的情况。第 3，该函数的参数是一个指向链表的指针，而不是一个链表。所以应该这样调用该函数：

```
InitializeList(&movies);
```

由于按值传递参数，所以该函数只能通过指向该变量的指针才能更改主调程序传入的变量。这里，由于语言的限制使得接口和抽象描述略有区别。

C 语言把所有类型和函数的信息集合成一个软件包的方法是：把类型定义和函数原型（包括前提条件和后置条件注释）放在一个头文件中。该文件应该提供程序员使用该类型所需的所有信息。程序清单 17.3 给出了一个简单链表类型的头文件。该程序定义了一个特定的结构作为 Item 类型，然后根据 Item 定义了 Node，再根据 Node 定义了 List。然后，把表示链表操作的函数设计为接受 Item 类型和 List 类型的参数。如果函数要修改一个参数，那么该参数的类型应是指向相应类型的指针，而不是该类型。在头文件中，把组成函数名的每个单词的首字母大写，以这种方式表明这些函数是接口包的一部分。另外，该文件使用第 16 章介绍的#ifndef 指令，防止多次包含一个文件。如果编译器不支持 C99 的 bool 类型，可以用下

面的代码：

```
enum bool {false, true}; /* 把 bool 定义为类型，false 和 true 是该类型的值 */
```

替换下面的头文件：

```
#include <stdbool.h> /* C99 特性 */
```

程序清单 17.3　`list.h` **接口头文件**

```
/* list.h -- 简单链表类型的头文件 */
#ifndef LIST_H_
#define LIST_H_
#include <stdbool.h>   /* C99 特性          */

/* 特定程序的声明 */

#define TSIZE      45   /* 储存电影名的数组大小   */
struct film
{
    char title[TSIZE];
    int rating;
};

/* 一般类型定义 */

typedef struct film Item;

typedef struct node
{
    Item item;
    struct node * next;
} Node;

typedef Node * List;

/* 函数原型 */

/* 操作：        初始化一个链表                                          */
/* 前提条件：    plist 指向一个链表                                      */
/* 后置条件：    链表初始化为空                                          */
void InitializeList(List * plist);

/* 操作：        确定链表是否为空定义，plist 指向一个已初始化的链表           */
/* 后置条件：    如果链表为空，该函数返回 true；否则返回 false               */
bool ListIsEmpty(const List *plist);

/* 操作：        确定链表是否已满，plist 指向一个已初始化的链表               */
/* 后置条件：    如果链表已满，该函数返回真；否则返回假                      */
bool ListIsFull(const List *plist);

/* 操作：        确定链表中的项数，plist 指向一个已初始化的链表               */
/* 后置条件：    该函数返回链表中的项数                                    */
unsigned int ListItemCount(const List *plist);
```

```
/* 操作：        在链表的末尾添加项                                              */
/* 前提条件：    item 是一个待添加至链表的项，plist 指向一个已初始化的链表       */
/* 后置条件：    如果可以，该函数在链表末尾添加一个项，且返回 true；否则返回 false */
bool AddItem(Item item, List * plist);

/* 操作：        把函数作用于链表中的每一项                                      */
/*               plist 指向一个已初始化的链表                                    */
/*               pfun 指向一个函数，该函数接受一个 Item 类型的参数，且无返回值   */
/* 后置条件：    pfun 指向的函数作用于链表中的每一项一次                         */
void Traverse(const List *plist, void(*pfun)(Item item));

/* 操作：        释放已分配的内存（如果有的话）                                  */
/*               plist 指向一个已初始化的链表                                    */
/* 后置条件：    释放了为链表分配的所有内存，链表设置为空                        */
void EmptyTheList(List * plist);

#endif
```

只有 InitializeList()、AddItem()和 EmptyTheList()函数要修改链表，因此从技术角度看，这些函数需要一个指针参数。然而，如果某些函数接受 List 类型的变量作为参数，而其他函数却接受 List 类型的地址作为参数，用户会很困惑。因此，为了减轻用户的负担，所有的函数均使用指针参数。

头文件中的一个函数原型比其他原型复杂：

```
/* 操作：        把函数作用于链表中的每一项                                    */
/*               plist 指向一个已初始化的链表                                  */
/*               pfun 指向一个函数，该函数接受一个 Item 类型的参数，且无返回值 */
/* 后置条件：    pfun 指向的函数作用于链表中的每一项一次                       */
void Traverse(const List *plist, void(*pfun)(Item item));
```

参数 pfun 是一个指向函数的指针，它指向的函数接受 item 值且无返回值。第 14 章中介绍过，可以把函数指针作为参数传递给另一个函数，然后该函数就可以使用这个被指针指向的函数。例如，该例中可以让 pfun 指向显示链表项的函数。然后把 Traverse()函数把该函数作用于链表中的每一项，显示链表中的内容。

17.3.3 使用接口

我们的目标是，使用这个接口编写程序，但是不必知道具体的实现细节（如，不知道函数的实现细节）。在编写具体函数之前，我们先编写电影程序的一个新版本。由于接口要使用 List 和 Item 类型，所以该程序也应使用这些类型。下面是编写该程序的一个伪代码方案。

```
创建一个 List 类型的变量。
创建一个 Item 类型的变量。
初始化链表为空。
当链表未满且有输入时:
    把输入读取到 Item 类型的变量中。
    在链表末尾添加项。
访问链表中的每个项并显示它们。
```

程序清单 17.4 中的程序按照以上伪代码来编写，其中还加入了一些错误检查。注意该程序利用了 list.h（程序清单 17.3）中描述的接口。另外，还需注意，链表中含有 showmovies()函数的代码，它与 Traverse()的原型一致。因此，程序可以把指针 showmovies 传递给 Traverse()，这样 Traverse()可以把 showmovies()函数应用于链表中的每一项（回忆一下，函数名是指向该函数的指针）。

程序清单 17.4　films3.c 程序

```
/* films3.c -- 使用抽象数据类型（ADT）风格的链表 */
/* 与 list.c 一起编译                         */
#include <stdio.h>
#include <stdlib.h>    /* 提供 exit()的原型 */
#include "list.h"      /* 定义 List、Item   */
void showmovies(Item item);
char * s_gets(char * st, int n);
int main(void)
{
    List movies;
    Item temp;


    /* 初始化      */
    InitializeList(&movies);
    if (ListIsFull(&movies))
    {
        fprintf(stderr, "No memory available! Bye!\n");
        exit(1);
    }

    /* 获取用户输入并储存 */
    puts("Enter first movie title:");
    while (s_gets(temp.title, TSIZE) != NULL && temp.title[0] != '\0')
    {
        puts("Enter your rating <0-10>:");
        scanf("%d", &temp.rating);
        while (getchar() != '\n')
            continue;
        if (AddItem(temp, &movies) == false)
        {
            fprintf(stderr, "Problem allocating memory\n");
            break;
        }
        if (ListIsFull(&movies))
        {
            puts("The list is now full.");
            break;
        }
        puts("Enter next movie title (empty line to stop):");
    }

    /* 显示         */
    if (ListIsEmpty(&movies))
        printf("No data entered. ");
    else
    {
```

```
        printf("Here is the movie list:\n");
        Traverse(&movies, showmovies);
    }
    printf("You entered %d movies.\n", ListItemCount(&movies));


    /* 清理        */
    EmptyTheList(&movies);
    printf("Bye!\n");

    return 0;
}

void showmovies(Item item)
{
    printf("Movie: %s  Rating: %d\n", item.title,
        item.rating);
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // 查找换行符
        if (find)                   // 如果地址不是 NULL,
            *find = '\0';           // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;           // 处理输入行的剩余内容
    }
    return ret_val;
}
```

17.3.4　实现接口

当然，我们还是必须实现 `List` 接口。C 方法是把函数定义统一放在 `list.c` 文件中。然后，整个程序由 `list.h`（定义数据结构和提供用户接口的原型）、`list.c`（提供函数代码实现接口）和 `films3.c`（把链表接口应用于特定编程问题的源代码文件）组成。程序清单 17.5 演示了 `list.c` 的一种实现。要运行该程序，必须把 `films3.c` 和 `list.c` 一起编译和链接（可以复习一下第 9 章关于编译多文件程序的内容）。`list.h`、`list.c` 和 `films3.c` 组成了整个程序（见图 17.5）。

程序清单 17.5　`list.c` 实现文件

```
/* list.c -- 支持链表操作的函数 */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"
```

```
/* 局部函数原型 */
static void CopyToNode(Item item, Node * pnode);

/* 接口函数  */
/* 把链表设置为空 */
void InitializeList(List * plist)
{
    *plist = NULL;
}

/* 如果链表为空，返回 true */
bool ListIsEmpty(const List * plist)
{
    if (*plist == NULL)
        return true;
    else
        return false;
}

/* 如果链表已满，返回 true */
bool ListIsFull(const List * plist)
{
    Node * pt;
    bool full;

    pt = (Node *)malloc(sizeof(Node));
    if (pt == NULL)
        full = true;
    else
        full = false;
    free(pt);

    return full;
}

/* 返回节点的数量 */
unsigned int ListItemCount(const List * plist)
{
    unsigned int count = 0;
    Node * pnode = *plist;    /* 设置链表的开始 */

    while (pnode != NULL)
    {
        ++count;
        pnode = pnode->next;  /* 设置下一个节点 */
    }

    return count;
}

/* 创建储存项的节点，并将其添加至由 plist 指向的链表末尾（较慢的实现） */
bool AddItem(Item item, List * plist)
{
    Node * pnew;
```

```
    Node * scan = *plist;

    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
        return false;     /* 失败时退出函数 */

    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (scan == NULL)          /* 空链表，所以把 */
        *plist = pnew;         /* pnew 放在链表的开头 */
    else
    {
        while (scan->next != NULL)
            scan = scan->next;  /* 找到链表的末尾 */
        scan->next = pnew;      /* 把 pnew 添加到链表的末尾 */
    }

    return true;
}

/* 访问每个节点并执行 pfun 指向的函数 */
void Traverse(const List * plist, void(*pfun)(Item item))
{
    Node * pnode = *plist;    /* 设置链表的开始 */

    while (pnode != NULL)
    {
        (*pfun)(pnode->item); /* 把函数应用于链表中的项 */
        pnode = pnode->next;  /* 前进到下一项   */
    }
}

/* 释放由 malloc()分配的内存 */
/* 设置链表指针为 NULL      */
void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next;     /* 保存下一个节点的地址   */
        free(*plist);               /* 释放当前节点           */
        *plist = psave;             /* 前进至下一个节点       */
    }
}

/* 局部函数定义 */
/* 把一个项拷贝到节点中 */
static void CopyToNode(Item item, Node * pnode)
{
    pnode->item = item;  /* 拷贝结构 */
}
```

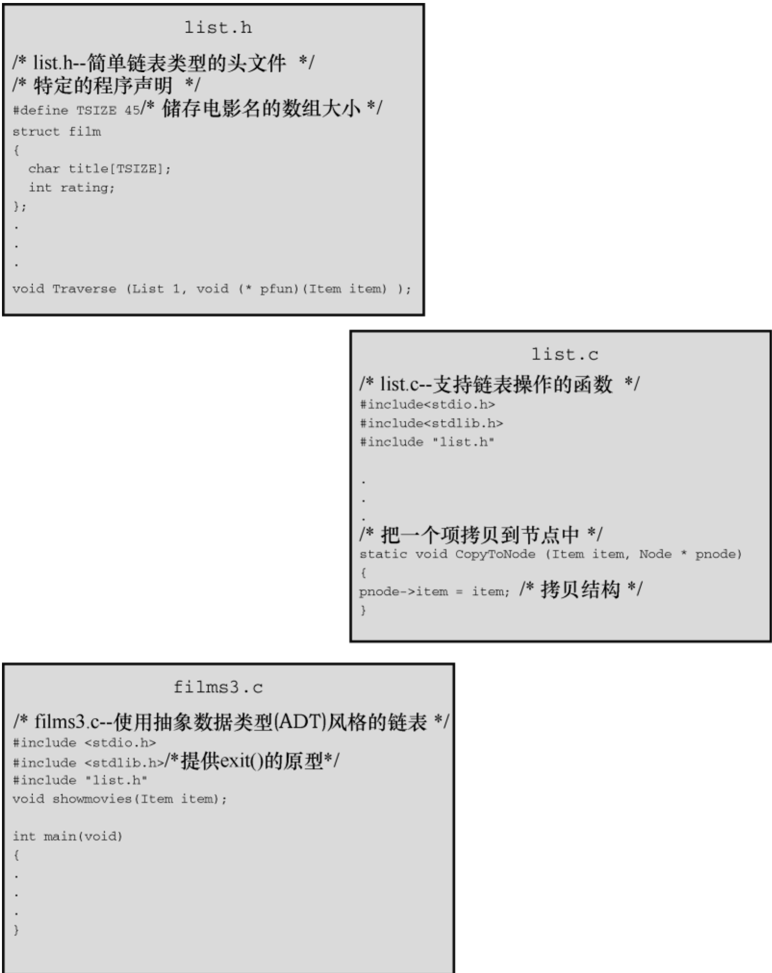


图 17.5　电影程序的 3 个部分

1. 程序的一些注释

list.c 文件有几个需要注意的地方。首先，该文件演示了什么情况下使用内部链接函数。如第 12 章所述，具有内部链接的函数只能在其声明所在的文件夹可见。在实现接口时，有时编写一个辅助函数（不作为正式接口的一部分）很方便。例如，使用 CopyToNode()函数把一个 Item 类型的值拷贝到 Item 类型的变量中。由于该函数是实现的一部分，但不是接口的一部分，所以我们使用 static 存储类别说明符把它隐藏在 list.c 文件中。接下来，讨论其他函数。

InitializeList()函数将链表初始化为空。在我们的实现中，这意味着把 List 类型的变量设置为 NULL。前面提到过，这要求把指向 List 类型变量的指针传递给该函数。

ListIsEmpty()函数很简单，但是它的前提条件是，当链表为空时，链表变量被设置为 NULL。因此，在首次调用 ListIsEmpty()函数之前初始化链表非常重要。另外，如果要扩展接口添加删除项的功能，那么当最后一个项被删除时，应该确保该删除函数重置链表为空。对链表而言，链表的大小取决于可用内存量。ListIsFull()函数尝试为新项分配空间。如果分配失败，说明链表已满；如果分配成功，则必须释放刚才分配的内存供真正的项所用。

ListItemCount()函数使用常用的链表算法遍历链表，同时统计链表中的项：

```
unsigned int ListItemCount(const List * plist)
{
    unsigned int count = 0;
    Node * pnode = *plist;    /* 设置链表的开始 */

    while (pnode != NULL)
    {
        ++count;
        pnode = pnode->next;  /* 设置下一个节点 */
    }

    return count;
}
```

AddItem()函数是这些函数中最复杂的：

```
bool AddItem(Item item, List * plist)
{
    Node * pnew;
    Node * scan = *plist;

    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
        return false;              /* 失败时退出函数 */

    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (scan == NULL)              /* 空链表，所以把 */
        *plist = pnew;             /* pnew 放在链表的开头 */
    else
    {
        while (scan->next != NULL)
            scan = scan->next; /* 找到链表的末尾 */
        scan->next = pnew;     /* 把 pnew 添加到链表的末尾 */
    }

    return true;
}
```

AddItem()函数首先为新节点分配空间。如果分配成功，该函数使用 CopyToNode()把项拷贝到新节点中。然后把该节点的 next 成员设置为 NULL。这表明该节点是链表中的最后一个节点。最后，完成创建节点并为其成员赋正确的值之后，该函数把该节点添加到链表的末尾。如果该项是添加到链表的第 1 个项，需要把头指针设置为指向第 1 项（记住，头指针的地址是传递给 AddItem()函数的第 2 个参数，所以*plist 就是头指针的值）。否则，代码继续在链表中前进，直到发现被设置为 NULL 的 next 成员。此时，该节点就是当前的最后一个节点，所以，函数重置它的 next 成员指向新节点。

要养成良好的编程习惯，给链表添加项之前应调用 ListIsFull()函数。但是，用户可能并未这样做，所以在 AddItem()函数内部检查 malloc()是否分配成功。而且，用户还可能在调用 ListIsFull()和调用 AddItem()函数之间做其他事情分配了内存，所以最好还是检查 malloc()是否分配成功。

Traverse()函数与 ListItemCount()函数类似，不过它还把一个指针函数作用于链表中的每一项。

```
void Traverse (const List * plist, void (* pfun)(Item item) )
{
    Node * pnode = *plist; /* 设置链表的开始 */
```

```
    while (pnode != NULL)
    {
         (*pfun)(pnode->item);  /* 把函数应用于该项*/
        pnode = pnode->next;    /* 前进至下一个项 */
    }
}
```

pnode->item 代表储存在节点中的数据，pnode->next 标识链表中的下一个节点。如下函数调用：

```
Traverse(movies, showmovies);
```

把 showmovies()函数应用于链表中的每一项。

最后，EmptyTheList()函数释放了之前 malloc()分配的内存：

```
void EmptyTheList(List * plist)
{
    Node * psave;

    while (*plist != NULL)
    {
        psave = (*plist)->next;     /* 保存下一个节点的地址   */
        free(*plist);               /* 释放当前节点           */
        *plist = psave;             /* 前进至下一个节点       */
    }
}
```

该函数的实现通过把 List 类型的变量设置为 NULL 来表明一个空链表。因此，要把 List 类型变量的地址传递给该函数，以便函数重置。由于 List 已经是一个指针，所以 plist 是一个指向指针的指针。因此，在上面的代码中，*plist 是指向 Node 的指针。当到达链表末尾时，*plist 为 NULL，表明原始的实际参数现在被设置为 NULL。

代码中要保存下一节点的地址，因为原则上调用了 free()会使当前节点（即*plist 指向的节点）的内容不可用。

提示　const 的限制

多个处理链表的函数都把 const List * plist 作为形参，表明这些函数不会更改链表。这里，const 确实提供了一些保护。它防止了*plist（即 plist 所指向的量）被修改。在该程序中，plist 指向 movies，所以 const 防止了这些函数修改 movies。因此，在 ListItemCount()中，不允许有类似下面的代码:

```
*plist = (*plist)->next; // 如果*plist 是 const，不允许这样做
```

因为改变*plist 就改变了 movies，将导致程序无法跟踪数据。然而，*plist 和 movies 都被看作是 const 并不意味着*plist 或 movies 指向的数据是 const。例如，可以编写下面的代码:

```
(*plist)->item.rating = 3; // 即使*plist 是 const，也可以这样做
```

因为上面的代码并未改变*plist，它改变的是*plist 指向的数据。由此可见，不要指望 const 能捕获到意外修改数据的程序错误。

2. 考虑你要做的

现在花点时间来评估 ADT 方法做了什么。首先，比较程序清单 17.2 和程序清单 17.4。这两个程序都使用相同的内存分配方法（动态分配链接的结构）解决电影链表的问题，但是程序清单 17.2 暴露了所有的编

程细节，把 malloc()和 prev->next 这样的代码都公之于众。而程序清单 17.4 隐藏了这些细节，并用与任务直接相关的方式表达程序。也就是说，该程序讨论的是创建链表和向链表中添加项，而不是调用内存函数或重置指针。简而言之，程序清单 17.4 是根据待解决的问题来表达程序，而不是根据解决问题所需的具体工具来表达程序。ADT 版本可读性更高，而且针对的是最终的用户所关心的问题。

其次，list.h 和 list.c 文件一起组成了可复用的资源。如果需要另一个简单的链表，也可以使用这些文件。假设你需要储存亲戚的一些信息：姓名、关系、地址和电话号码，那么先要在 list.h 文件中重新定义 Item 类型：

```
typedef struct itemtag
{
    char fname[14];
    char lname [24];
    char relationship[36];
    char address [60];
    char phonenum[20];
} Item;
```

然后……只需要做这些就行了。因为所有处理简单链表的函数都与 Item 类型有关。根据不同的情况，有时还要重新定义 CopyToNode()函数。例如，当项是一个数组时，就不能通过赋值来拷贝。

另一个要点是，用户接口是根据抽象链表操作定义的，不是根据某些特定的数据表示和算法来定义。这样，不用重写最后的程序就能随意修改实现。例如，当前使用的 AddItem()函数效率不高，因为它总是从链表第 1 个项开始，然后搜索至链表末尾。可以通过保存链表结尾处的地址来解决这个问题。例如，可以这样重新定义 List 类型：

```
typedef struct list
{
    Node * head;  /* 指向链表的开头 */
    Node * end;   /* 指向链表的末尾 */
} List;
```

当然，还要根据新的定义重写处理链表的函数，但是不用修改程序清单 17.4 中的内容。对大型编程项目而言，这种把实现和最终接口隔离的做法相当有用。这称为数据隐藏，因为对终端用户隐藏了数据表示的细节。

注意，这种特殊的 ADT 甚至不要求以链表的方式实现简单链表。下面是另一种方法：

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE]; /* 项数组 */
    int items;                /* 其中的项数 */
} List;
```

这样做也需要重写 list.c 文件，但是使用 list 的程序不用修改。

最后，考虑这种方法给程序开发过程带来了哪些好处。如果程序运行出现问题，可以把问题定位到具体的函数上。如果想用更好的方法来完成某个任务（如，添加项），只需重写相应的函数即可。如果需要新功能，可以添加一个新的函数。如果觉得数组或双向链表更好，可以重写实现的代码，不用修改使用实现的程序。

17.4　队列 ADT

在 C 语言中使用抽象数据类型方法编程包含以下 3 个步骤。

1．以抽象、通用的方式描述一个类型，包括该类型的操作。

2．设计一个函数接口表示这个新类型。

3．编写具体代码实现这个接口。

前面已经把这种方法应用到简单链表中。现在，把这种方法应用于更复杂的数据类型：队列。

17.4.1　定义队列抽象数据类型

队列（*queue*）是具有两个特殊属性的链表。第一，新项只能添加到链表的末尾。从这方面看，队列与简单链表类似。第二，只能从链表的开头移除项。可以把队列想象成排队买票的人。你从队尾加入队列，买完票后从队首离开。队列是一种“先进先出”（*first in , first out*，缩写为 FIFO）的数据形式，就像排队买票的队伍一样（前提是没有人插队）。接下来，我们建立一个非正式的抽象定义：

类型名：　　　　队列

类型属性：　　　可以储存一系列项

类型操作：　　　初始化队列为空

　　　　　　　　　　确定队列为空

　　　　　　　　　　确定队列已满

　　　　　　　　　　确定队列中的项数

　　　　　　　　　　在队列末尾添加项

　　　　　　　　　　在队列开头删除或恢复项

　　　　　　　　　　清空队列

17.4.2　定义一个接口

接口定义放在 `queue.h` 文件中。我们使用 C 的 `typedef` 工具创建两个类型名：`Item` 和 `Queue`。相应结构的具体实现应该是 `queue.h` 文件的一部分，但是从概念上来看，应该在实现阶段才设计结构。现在，只是假定已经定义了这些类型，着重考虑函数的原型。

首先，考虑初始化。这涉及改变 `Queue` 类型，所以该函数应该以 `Queue` 的地址作为参数：

```
void InitializeQueue (Queue * pq);
```

接下来，确定队列是否为空或已满的函数应返回真或假值。这里，假设 C99 的 `stdbool.h` 头文件可用。如果该文件不可用，可以使用 `int` 类型或自己定义 `bool` 类型。由于该函数不更改队列，所以接受 `Queue` 类型的参数。但是，传递 `Queue` 的地址更快，更节省内存，这取决于 `Queue` 类型的对象大小。这次我们尝试这种方法。这样做的好处是，所有的函数都以地址作为参数，而不像 `List` 示例那样。为了表明这些函数不更改队列，可以且应该使用 `const` 限定符：

```
bool QueueIsFull(const Queue * pq);
bool QueueIsEmpty (const Queue * pq);
```

指针 `pq` 指向 `Queue` 数据对象，不能通过 `pq` 这个代理更改数据。可以定义一个类似该函数的原型，返回队列的项数：

```
int QueueItemCount(const Queue * pq);
```

在队列末尾添加项涉及标识项和队列。这次要更改队列，所以有必要（而不是可选）使用指针。该函数的返回类型可以是 `void`，或者通过返回值来表示是否成功添加项。我们采用后者：

```
bool EnQueue(Item item, Queue * pq);
```

最后，删除项有多种方法。如果把项定义为结构或一种基本类型，可以通过函数返回待删除的项。函

数的参数可以是 Queue 类型或指向 Queue 的指针。因此，可能是下面这样的原型：

```
Item DeQueue(Queue q);
```

然而，下面的原型会更合适一些：

```
bool DeQueue(Item * pitem, Queue * pq);
```

从队列中待删除的项储存在 pitem 指针指向的位置，函数的返回值表明是否删除成功。

清空队列的函数所需的唯一参数是队列的地址，可以使用下面的函数原型：

```
void EmptyTheQueue(Queue * pq);
```

17.4.3　实现接口数据表示

第一步是确定在队列中使用何种 C 数据形式。有可能是数组。数组的优点是方便使用，而且向数组的末尾添加项很简单。问题是如何从队列的开头删除项。类比于排队买票的队列，从队列的开头删除一个项包括拷贝数组首元素的值和把数组剩余各项依次向前移动一个位置。编程实现这个过程很简单，但是会浪费大量的计算机时间（见图 17.6）。

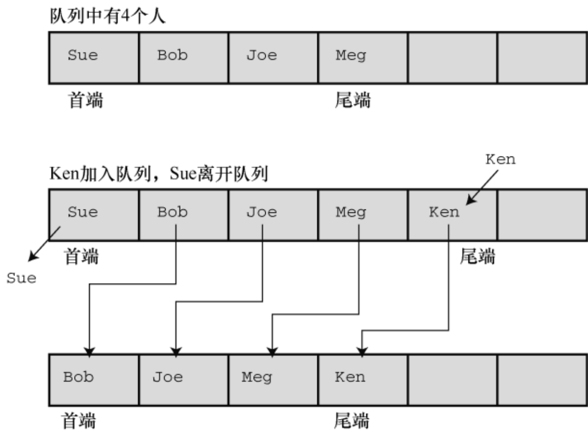


图 17.6　用数组实现队列

第二种解决数组队列删除问题的方法是改变队列首端的位置，其余元素不动（见图 17.7）。

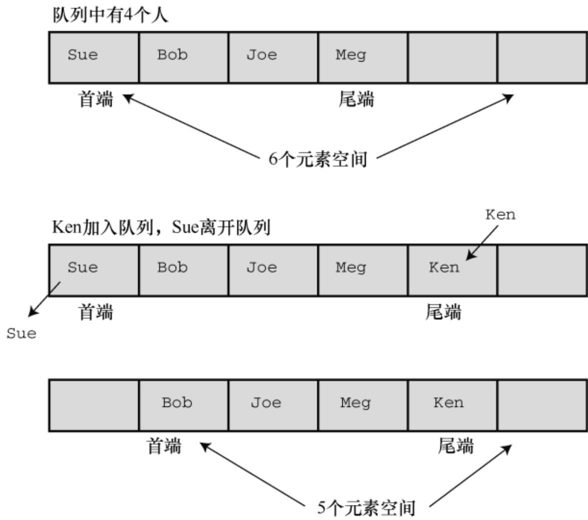


图 17.7　重新定义首元素

解决这种问题的一个好方法是，使队列成为环形。这意味着把数组的首尾相连，即数组的首元素紧跟在最后一个元素后面。这样，当到达数组末尾时，如果首元素空出，就可以把新添加的项储存到这些空出的元素中（见图 17.8）。可以想象在一张条形的纸上画出数组，然后把数组的首尾粘起来形成一个环。当然，要做一些标记，以免尾端超过首端。

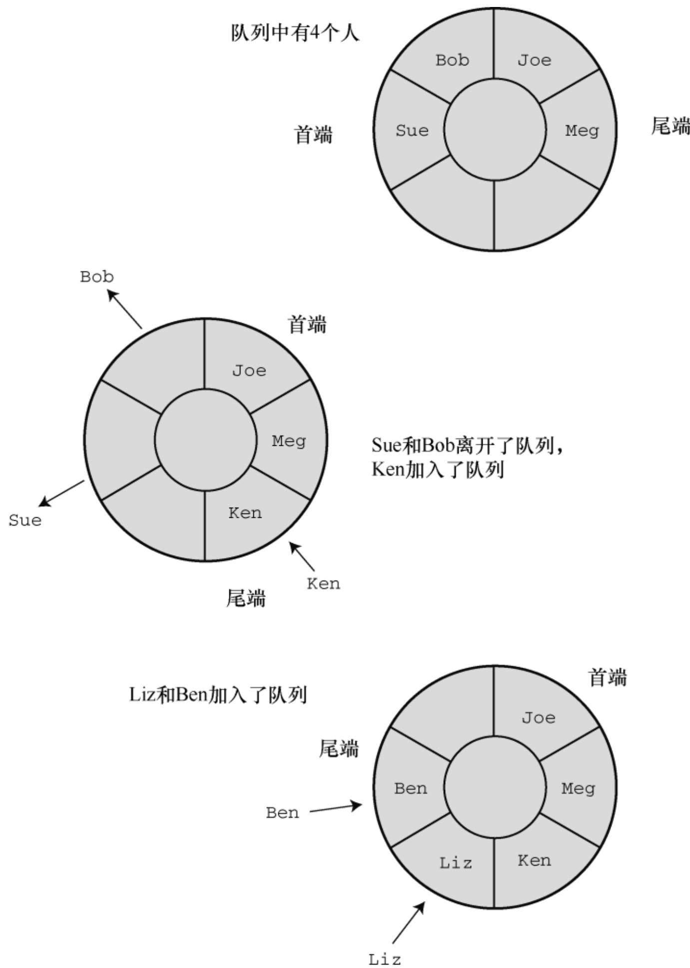


图 17.8　环形队列

另一种方法是使用链表。使用链表的好处是删除首项时不必移动其余元素，只需重置头指针指向新的首元素即可。由于我们已经讨论过链表，所以采用这个方案。我们用一个整数队列开始测试：

```
typedef int Item;
```

链表由节点组成，所以，下一步是定义节点：

```
typedef struct node
{
    Item item;
    struct node * next;
} Node;
```

对队列而言，要保存首尾项，这可以使用指针来完成。另外，可以用一个计数器来记录队列中的项数。因此，该结构应由两个指针成员和一个 int 类型的成员构成：

```
typedef struct queue
{
    Node * front; /* 指向队列首项的指针 */
    Node * rear;  /*指向队列尾项的指针*/
    int items;         /* 队列中的项数*/
} Queue;
```

注意，Queue 是一个内含 3 个成员的结构，所以用指向队列的指针作为参数比直接用队列作为参数节约了时间和空间。

接下来，考虑队列的大小。对链表而言，其大小受限于可用的内存量，因此链表不要太大。例如，可能使用一个队列模拟飞机等待在机场着陆。如果等待的飞机数量太多，新到的飞机就应该改到其他机场降落。我们把队列的最大长度设置为 10。程序清单 17.6 包含了队列接口的原型和定义。Item 类型留给用户定义。使用该接口时，可以根据特定的程序插入合适的定义。

程序清单 17.6　queue.h 接口头文件

```
/* queue.h -- Queue 的接口 */
#ifndef _QUEUE_H_
#define _QUEUE_H_
#include <stdbool.h>

// 在这里插入 Item 类型的定义，例如
typedef int Item;      // 用于 use_q.c
// 或者 typedef struct item {int gumption; int charisma;} Item;

#define MAXQUEUE 10

typedef struct node
{
    Item item;
    struct node * next;
} Node;

typedef struct queue
{
    Node * front;  /* 指向队列首项的指针    */
    Node * rear;   /* 指向队列尾项的指针    */
    int items;     /* 队列中的项数          */
} Queue;

/* 操作:        初始化队列                                  */
/* 前提条件:    pq 指向一个队列                             */
/* 后置条件:    队列被初始化为空                            */
void InitializeQueue(Queue * pq);

/* 操作:        检查队列是否已满                            */
/* 前提条件:    pq 指向之前被初始化的队列                   */
/* 后置条件:    如果队列已满则返回 true，否则返回 false     */
bool QueueIsFull(const Queue * pq);

/* 操作:        检查队列是否为空                            */
/* 前提条件:    pq 指向之前被初始化的队列                   */
/* 后置条件:    如果队列为空则返回 true，否则返回 false     */
```

```
bool QueueIsEmpty(const Queue *pq);

/* 操作：          确定队列中的项数                                    */
/* 前提条件：      pq 指向之前被初始化的队列                           */
/* 后置条件：      返回队列中的项数                                    */
int QueueItemCount(const Queue * pq);

/* 操作：          在队列末尾添加项                                    */
/* 前提条件：      pq 指向之前被初始化的队列                           */
/*                 item 是要被添加在队列末尾的项                       */
/* 后置条件：      如果队列不为空，item 将被添加在队列的末尾，         */
/*                 该函数返回 true；否则，队列不改变，该函数返回 false */
bool EnQueue(Item item, Queue * pq);

/* 操作：          从队列的开头删除项                                  */
/* 前提条件：      pq 指向之前被初始化的队列                           */
/* 后置条件：      如果队列不为空，队列首端的 item 将被拷贝到*pitem 中 */
/*                 并被删除，且函数返回 true；                         */
/*                 如果该操作使得队列为空，则重置队列为空              */
/*                 如果队列在操作前为空，该函数返回 false              */
bool DeQueue(Item *pitem, Queue * pq);

/* 操作：          清空队列                                            */
/* 前提条件：      pq 指向之前被初始化的队列                           */
/* 后置条件：      队列被清空                                          */
void EmptyTheQueue(Queue * pq);

#endif
```

1. 实现接口函数

接下来，我们编写接口代码。首先，初始化队列为空，这里“空”的意思是把指向队列首项和尾项的指针设置为 NULL，并把项数（items 成员）设置为 0：

```
void InitializeQueue(Queue * pq)
{
    pq->front = pq->rear = NULL;
    pq->items = 0;
}
```

这样，通过检查 items 的值可以很方便地了解到队列是否已满、是否为空和确定队列的项数：

```
bool QueueIsFull(const Queue * pq)
{
    return pq->items == MAXQUEUE;
}
bool QueueIsEmpty(const Queue * pq)
{
    return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
    return pq->items;
}
```

把项添加到队列中，包括以下几个步骤：

（1）创建一个新节点；

（2）把项拷贝到节点中；

（3）设置节点的 next 指针为 NULL，表明该节点是最后一个节点；

（4）设置当前尾节点的 next 指针指向新节点，把新节点链接到队列中；

（5）把 rear 指针指向新节点，以便找到最后的节点；

（6）项数加 1。

函数还要处理两种特殊情况。第一种情况，如果队列为空，应该把 front 指针设置为指向新节点。因为如果队列中只有一个节点，那么这个节点既是首节点也是尾节点。第二种情况是，如果函数不能为节点分配所需内存，则必须执行一些动作。因为大多数情况下我们都使用小型队列，这种情况很少发生，所以，如果程序运行的内存不足，我们只是通过函数终止程序。EnQueue()的代码如下：

```
bool EnQueue(Item item, Queue * pq)
{
    Node * pnew;

    if (QueueIsFull(pq))
        return false;
    pnew = (Node *)malloc( sizeof(Node));
    if (pnew == NULL)
    {
        fprintf(stderr,"Unable to allocate memory!\n");
        exit(1);
    }
    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (QueueIsEmpty(pq))
        pq->front = pnew;       /* 项位于队列首端         */
    else
        pq->rear->next = pnew; /* 链接到队列尾端        */
    pq->rear = pnew;            /* 记录队列尾端的位置     */
    pq->items++;                /* 队列项数加 1           */

    return true;
}
```

CopyToNode()函数是静态函数，用于把项拷贝到节点中：

```
static void CopyToNode(Item item, Node * pn)
{
    pn->item = item;
}
```

从队列的首端删除项，涉及以下几个步骤：

（1）把项拷贝到给定的变量中；

（2）释放空出的节点使用的内存空间；

（3）重置首指针指向队列中的下一个项；

（4）如果删除最后一项，把首指针和尾指针都重置为 NULL；

（5）项数减 1。

下面的代码完成了这些步骤：

```
bool DeQueue(Item * pitem, Queue * pq)
{
    Node * pt;

    if (QueueIsEmpty(pq))
        return false;
    CopyToItem(pq->front, pitem);
    pt = pq->front;
    pq->front = pq->front->next;
    free(pt);
    pq->items--;
    if (pq->items == 0)
        pq->rear = NULL;

    return true;
}
```

关于指针要注意两点。第一，删除最后一项时，代码中并未显式设置 front 指针为 NULL，因为已经设置 front 指针指向被删除节点的 next 指针。如果该节点不是最后一个节点，那么它的 next 指针就为 NULL。第二，代码使用临时指针（pt）储存待删除节点的位置。因为指向首节点的正式指针（pt->front）被重置为指向下一个节点，所以如果没有临时指针，程序就不知道该释放哪块内存。

我们使用 DeQueue() 函数清空队列。循环调用 DeQueue() 函数直到队列为空：

```
void EmptyTheQueue(Queue * pq)
{
    Item dummy;
    while (!QueueIsEmpty(pq))
        DeQueue(&dummy, pq);
}
```

注意　保持纯正的 ADT

定义 ADT 接口后，应该只使用接口函数处理数据类型。例如，Dequeue()依赖 EnQueue()函数来正确设置指针和把 rear 节点的 next 指针设置为 NULL。如果在一个使用 ADT 的程序中，决定直接操控队列的某些部分，有可能破坏接口包中函数之间的协作关系。

程序清单 17.7 演示了该接口中的所有函数，包括 EnQueue()函数中用到的 CopyToItem()函数。

程序清单 17.7　queue.c 实现文件

```
/* queue.c -- Queue 类型的实现 */
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

/* 局部函数 */
static void CopyToNode(Item item, Node * pn);
static void CopyToItem(Node * pn, Item * pi);

void InitializeQueue(Queue * pq)
{
    pq->front = pq->rear = NULL;
```

```
    pq->items = 0;
}

bool QueueIsFull(const Queue * pq)
{
    return pq->items == MAXQUEUE;
}

bool QueueIsEmpty(const Queue * pq)
{
    return pq->items == 0;
}

int QueueItemCount(const Queue * pq)
{
    return pq->items;
}

bool EnQueue(Item item, Queue * pq)
{
    Node * pnew;

    if (QueueIsFull(pq))
        return false;
    pnew = (Node *) malloc(sizeof(Node));
    if (pnew == NULL)
    {
        fprintf(stderr, "Unable to allocate memory!\n");
        exit(1);
    }
    CopyToNode(item, pnew);
    pnew->next = NULL;
    if (QueueIsEmpty(pq))
        pq->front = pnew;           /* 项位于队列的首端       */
    else
        pq->rear->next = pnew;      /* 链接到队列的尾端       */
    pq->rear = pnew;                /* 记录队列尾端的位置     */
    pq->items++;                    /* 队列项数加 1           */

    return true;
}

bool DeQueue(Item * pitem, Queue * pq)
{
    Node * pt;

    if (QueueIsEmpty(pq))
        return false;
    CopyToItem(pq->front, pitem);
    pt = pq->front;
    pq->front = pq->front->next;
    free(pt);
    pq->items--;
    if (pq->items == 0)
        pq->rear = NULL;
```

```
    return true;
}

/* 清空队列 */
void EmptyTheQueue(Queue * pq)
{
    Item dummy;
    while (!QueueIsEmpty(pq))
        DeQueue(&dummy, pq);
}

/* 局部函数 */

static void CopyToNode(Item item, Node * pn)
{
    pn->item = item;
}

static void CopyToItem(Node * pn, Item * pi)
{
    *pi = pn->item;
}
```

17.4.4　测试队列

在重要程序中使用一个新的设计（如，队列包）之前，应该先测试该设计。测试的一种方法是，编写一个小程序。这样的程序称为*驱动程序*（*driver*），其唯一的用途是进行测试。例如，程序清单 17.8 使用一个添加和删除整数的队列。在运行该程序之前，要确保 `queue.h` 中包含下面这行代码：

```
typedef int item;
```

记住，还必须链接 `queue.c` 和 `use_q.c`。

程序清单 17.8　`use_q.c` 程序

```
/* use_q.c -- 驱动程序测试 Queue 接口   */
/* 与 queue.c 一起编译                   */
#include <stdio.h>
#include "queue.h"  /* 定义 Queue、Item   */

int main(void)
{
    Queue line;
    Item temp;
    char ch;

    InitializeQueue(&line);
    puts("Testing the Queue interface. Type a to add a value,");
    puts("type d to delete a value, and type q to quit.");
    while ((ch = getchar()) != 'q')
    {
        if (ch != 'a' && ch != 'd')   /* 忽略其他输出 */
            continue;
```

```
        if (ch == 'a')
        {
            printf("Integer to add: ");
            scanf("%d", &temp);
            if (!QueueIsFull(&line))
            {
                printf("Putting %d into queue\n", temp);
                EnQueue(temp, &line);
            }
            else
                puts("Queue is full!");
        }
        else
        {
            if (QueueIsEmpty(&line))
                puts("Nothing to delete!");
            else
            {
                DeQueue(&temp, &line);
                printf("Removing %d from queue\n", temp);
            }
        }
        printf("%d items in queue\n", QueueItemCount(&line));
        puts("Type a to add, d to delete, q to quit:");
    }
    EmptyTheQueue(&line);
    puts("Bye!");

    return 0;
}
```

下面是一个运行示例。除了这样测试，还应该测试当队列已满后，实现是否能正常运行。

```
Testing the Queue interface. Type a to add a value,
type d to delete a value, and type q to quit.
a
Integer to add: 40
Putting 40 into queue
1 items in queue
Type a to add, d to delete, q to quit:
a
Integer to add: 20
Putting 20 into queue
2 items in queue
Type a to add, d to delete, q to quit:
a
Integer to add: 55
Putting 55 into queue
3 items in queue
Type a to add, d to delete, q to quit:
d
Removing 40 from queue
2 items in queue
Type a to add, d to delete, q to quit:
d
Removing 20 from queue
```

```
1 items in queue
Type a to add, d to delete, q to quit:
d
Removing 55 from queue
0 items in queue
Type a to add, d to delete, q to quit:
d
Nothing to delete!
0 items in queue
Type a to add, d to delete, q to quit:
q
Bye!
```

17.5　用队列进行模拟

经过测试，队列没问题。现在，我们用它来做一些有趣的事情。许多现实生活的情形都涉及队列。例如，在银行或超市的顾客队列、机场的飞机队列、多任务计算机系统中的任务队列等。我们可以用队列包来模拟这些情形。

假设 Sigmund Landers 在商业街设置了一个提供建议的摊位。顾客可以购买 1 分钟、2 分钟或 3 分钟的建议。为确保交通畅通，商业街规定每个摊位前排队等待的顾客最多为 10 人（相当于程序中的最大队列长度）。假设顾客都是随机出现的，并且他们花在咨询上的时间也是随机选择的（1 分钟、2 分钟、3 分钟）。那么 Sigmund 平均每小时要接待多少名顾客？每位顾客平均要花多长时间？排队等待的顾客平均有多少人？队列模拟能回答类似的问题。

首先，要确定在队列中放什么。可以根据顾客加入队列的时间和顾客咨询时花费的时间来描述每一位顾客。因此，可以这样定义 Item 类型。

```
typedef struct item
{
    long arrive;       /* 一位顾客加入队列的时间 */
    int processtime;   /* 该顾客咨询时花费的时间 */
} Item;
```

要用队列包来处理这个结构，必须用 typedef 定义的 Item 替换上一个示例的 int 类型。这样做就不用担心队列的具体工作机制，可以集中精力分析实际问题，即模拟咨询 Sigmund 的顾客队列。

这里有一种方法，让时间以 1 分钟为单位递增。每递增 1 分钟，就检查是否有新顾客到来。如果有一位顾客且队列未满，将该顾客添加到队列中。这涉及把顾客到来的时间和顾客所需的咨询时间记录在 Item 类型的结构中，然后在队列中添加该项。然而，如果队列已满，就让这位顾客离开。为了做统计，要记录顾客的总数和被拒顾客（队列已满不能加入队列的人）的总数。

接下来，处理队列的首端。也就是说，如果队列不为空且前面的顾客没有在咨询 Sigmund，则删除队列首端的项。记住，该项中储存着这位顾客加入队列的时间，把该时间与当前时间作比较，就可得出该顾客在队列中等待的时间。该项还储存着这位顾客需要咨询的分钟数，即还要咨询 Sigmund 多长时间。因此还要用一个变量储存这个时长。如果 Sigmund 正忙，则不用让任何人离开队列。尽管如此，记录等待时间的变量应该递减 1。

核心代码类似下面这样，每一轮迭代对应 1 分钟的行为：

```
for (cycle = 0; cycle < cyclelimit; cycle++)
{
    if (newcustomer(min_per_cust))
    {
```

```
            if (QueueIsFull(&line))
                turnaways++;
            else
            {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
            }
        }
        if (wait_time <= 0 && !QueueIsEmpty(&line))
        {
            DeQueue(&temp, &line);
            wait_time = temp.processtime;
            line_wait += cycle - temp.arrive;
            served++;
        }
        if (wait_time > 0)
            wait_time--;
        sum_line += QueueItemCount(&line);
    }
```

注意，时间的表示比较粗糙（1 分钟），所以一小时最多 60 位顾客。下面是一些变量和函数的含义。

- min_per_cus 是顾客到达的平均间隔时间。
- newcustomer()使用 C 的 rand()函数确定在特定时间内是否有顾客到来。
- turnaways 是被拒绝的顾客数量。
- customers 是加入队列的顾客数量。
- temp 是表示新顾客的 Item 类型变量。
- customertime()设置 temp 结构中的 arrive 和 processtime 成员。
- wait_time 是 Sigmund 完成当前顾客的咨询还需多长时间。
- line_wait 是到目前为止队列中所有顾客的等待总时间。
- served 是咨询过 Sigmund 的顾客数量。
- sum_line 是到目前为止统计的队列长度。

如果到处都是 malloc()、free()和指向节点的指针，整个程序代码会非常混乱和晦涩。队列包让你把注意力集中在模拟问题上，而不是编程细节上。

程序清单 17.9 演示了模拟商业街咨询摊位队列的完整代码。根据第 12 章介绍的方法，使用标准函数 rand()、srand()和 time()来产生随机数。另外要特别注意，必须用下面的代码更新 queue.h 中的 Item，该程序才能正常工作：

```
typedef struct item
{
    long arrive;        //一位顾客加入队列的时间
    int processtime;    //该顾客咨询时花费的时间
} Item;
```

记住，还要把 mall.c 和 queue.c 一起链接。

程序清单 17.9 mall.c 程序

```
// mall.c -- 使用 Queue 接口
// 和 queue.c 一起编译
```

```
#include <stdio.h>
#include <stdlib.h>            // 提供 rand() 和 srand() 的原型
#include <time.h>              // 提供 time() 的原型
#include "queue.h"             // 更改 Item 的 typedef
#define MIN_PER_HR 60.0

bool newcustomer(double x);    // 是否有新顾客到来?
Item customertime(long when);  // 设置顾客参数

int main(void)
{
    Queue line;
    Item temp;                 // 新的顾客数据
    int hours;                 // 模拟的小时数
    int perhour;               // 每小时平均多少位顾客
    long cycle, cyclelimit;    // 循环计数器、计数器的上限
    long turnaways = 0;        // 因队列已满被拒的顾客数量
    long customers = 0;        // 加入队列的顾客数量
    long served = 0;           // 在模拟期间咨询过 Sigmund 的顾客数量
    long sum_line = 0;         // 累计的队列总长
    int wait_time = 0;         // 从当前到 Sigmund 空闲所需的时间
    double min_per_cust;       // 顾客到来的平均时间
    long line_wait = 0;        // 队列累计的等待时间

    InitializeQueue(&line);
    srand((unsigned int) time(0)); // rand() 随机初始化
    puts("Case Study: Sigmund Lander's Advice Booth");
    puts("Enter the number of simulation hours:");
    scanf("%d", &hours);
    cyclelimit = MIN_PER_HR * hours;
    puts("Enter the average number of customers per hour:");
    scanf("%d", &perhour);
    min_per_cust = MIN_PER_HR / perhour;

    for (cycle = 0; cycle < cyclelimit; cycle++)
    {
        if (newcustomer(min_per_cust))
        {
            if (QueueIsFull(&line))
                turnaways++;
            else
            {
                customers++;
                temp = customertime(cycle);
                EnQueue(temp, &line);
            }
        }
        if (wait_time <= 0 && !QueueIsEmpty(&line))
        {
            DeQueue(&temp, &line);
            wait_time = temp.processtime;
            line_wait += cycle - temp.arrive;
```

```
                served++;
            }
            if (wait_time > 0)
                wait_time--;
            sum_line += QueueItemCount(&line);
        }

        if (customers > 0)
        {
            printf("customers accepted: %ld\n", customers);
            printf("  customers served: %ld\n", served);
            printf("       turnaways: %ld\n", turnaways);
            printf("average queue size: %.2f\n",
                (double) sum_line / cyclelimit);
            printf(" average wait time: %.2f minutes\n",
                (double) line_wait / served);
        }
        else
            puts("No customers!");
        EmptyTheQueue(&line);
        puts("Bye!");

        return 0;
    }

    // x是顾客到来的平均时间（单位：分钟）
    // 如果1分钟内有顾客到来，则返回true
    bool newcustomer(double x)
    {
        if (rand() * x / RAND_MAX < 1)
            return true;
        else
            return false;
    }

    // when是顾客到来的时间
    // 该函数返回一个Item结构，该顾客到达的时间设置为when,
    // 咨询时间设置为1～3的随机值
    Item customertime(long when)
    {
        Item cust;

        cust.processtime = rand() % 3 + 1;
        cust.arrive = when;

        return cust;
    }
```

该程序允许用户指定模拟运行的小时数和每小时平均有多少位顾客。模拟时间较长得出的值较为平均，模拟时间较短得出的值随时间的变化而随机变化。下面的运行示例解释了这一点（先保持每小时的顾客平均数量不变）。注意，在模拟 80 小时和 800 小时的情况下，平均队伍长度和等待时间基本相同。但是，在模拟 1 小时的情况下这两个量差别很大，而且与长时间模拟的情况差别也很大。这是因为小数量的统计样本往往更容易受相对变化的影响。

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
20
customers accepted: 1633
  customers served: 1633
         turnaways: 0
average queue size: 0.46
average wait time: 1.35 minutes

Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
800
Enter the average number of customers per hour:
20
customers accepted: 16020
  customers served: 16019
         turnaways: 0
average queue size: 0.44
average wait time: 1.32 minutes

Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
1
Enter the average number of customers per hour:
20
customers accepted: 20
  customers served: 20
         turnaways: 0
average queue size: 0.23
average wait time: 0.70 minutes

Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
1
Enter the average number of customers per hour:
20
customers accepted: 22
  customers served: 22
         turnaways: 0
average queue size: 0.75
average wait time: 2.05 minutes
```

然后保持模拟的时间不变，改变每小时的顾客平均数量：

```
Case Study: Sigmund Lander's Advice Booth
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
25
customers accepted: 1960
  customers served: 1959
         turnaways: 3
average queue size: 1.43
average wait time: 3.50 minutes

Case Study: Sigmund Lander's Advice Booth
```

```
Enter the number of simulation hours:
80
Enter the average number of customers per hour:
30
customers accepted: 2376
  customers served: 2373
         turnaways: 94
average queue size: 5.85
average wait time: 11.83 minutes
```

注意，随着每小时顾客平均数量的增加，顾客的平均等待时间迅速增加。在每小时 20 位顾客（80 小时模拟时间）的情况下，每位顾客的平均等待时间是 1.35 分钟；在每小时 25 位顾客的情况下，平均等待时间增加至 3.50 分钟；在每小时 30 位顾客的情况下，该数值攀升至 11.83 分钟。而且，这 3 种情况下被拒顾客分别从 0 位增加至 3 位最后陡增至 94 位。Sigmund 可以根据程序模拟的结果决定是否要增加一个摊位。

17.6 链表和数组

许多编程问题，如创建一个简单链表或队列，都可以用链表（指的是动态分配结构的序列链）或数组来处理。每种形式都有其优缺点，所以要根据具体问题的要求来决定选择哪一种形式。表 17.1 总结了链表和数组的性质。

表 17.1 比较数组和链表

数据形式	优点	缺点
数组	C 直接支持 提供随机访问	在编译时确定大小 插入和删除元素很费时
链表	运行时确定大小 快速插入和删除元素	不能随机访问 用户必须提供编程支持

接下来，详细分析插入和删除元素的过程。在数组中插入元素，必须移动其他元素腾出空位插入新元素，如图 17.9 所示。新插入的元素离数组开头越近，要被移动的元素越多。然而，在链表中插入节点，只需给两个指针赋值，如图 17.10 所示。类似地，从数组中删除一个元素，也要移动许多相关的元素。但是从链表中删除节点，只需重新设置一个指针并释放被删除节点占用的内存即可。

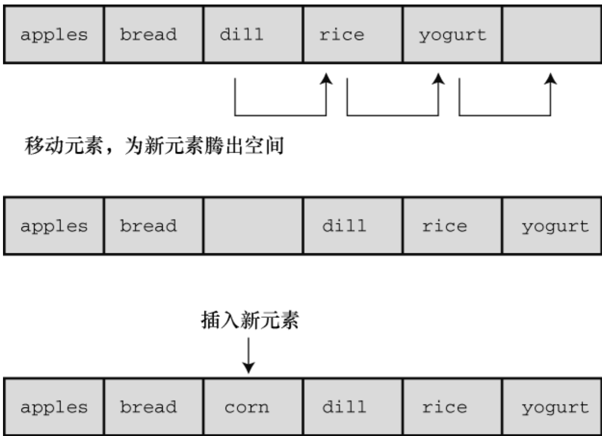


图 17.9 在数组中插入一个元素

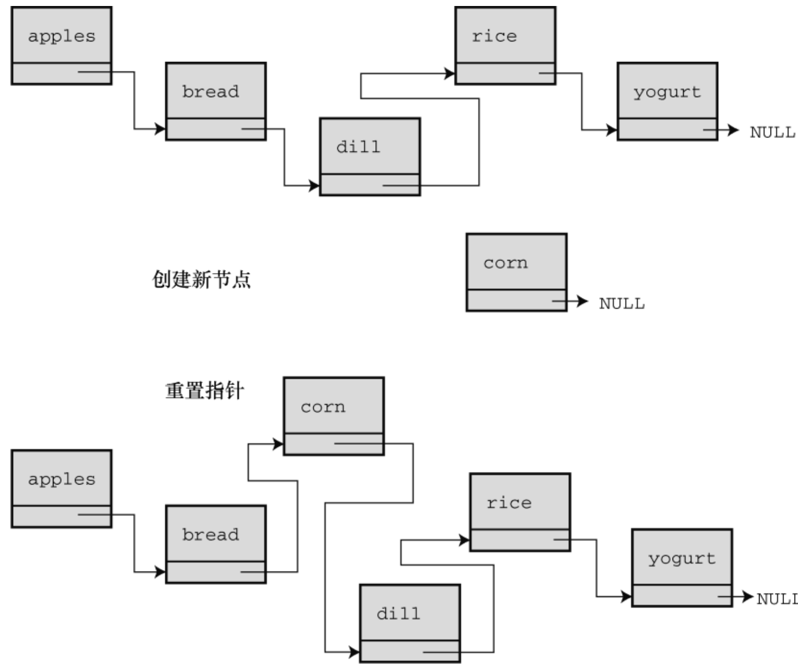


图 17.10　在链表中插入一个元素

接下来，考虑如何访问元素。对数组而言，可以使用数组下标直接访问该数组中的任意元素，这叫做随机访问（*random access*）。对链表而言，必须从链表首节点开始，逐个节点移动到要访问的节点，这叫做顺序访问（*sequential access*）。当然，也可以顺序访问数组。只需按顺序递增数组下标即可。在某些情况下，顺序访问足够了。例如，显示链表中的每一项，顺序访问就不错。其他情况用随机访问更合适。

假设要查找链表中的特定项。一种算法是从列表的开头开始按顺序查找，这叫做顺序查找（*sequential search*）。如果项并未按某种顺序排列，则只能顺序查找。如果待查找的项不在链表里，必须查找完所有的项才知道该项不在链表中（在这种情况下可以使用并发编程，同时查找列表中的不同部分）。

我们可以先排序列表，以改进顺序查找。这样，就不必查找排在待查找项后面的项。例如，假设在一个按字母排序的列表中查找 `Susan`。从开头开始查找每一项，直到 `Sylvia` 都没有查找到 `Susan`。这时就可以退出查找，因为如果 `Susan` 在列表中，应该排在 `Sylvia` 前面。平均下来，这种方法查找不在列表中的项的时间减半。

对于一个排序的列表，用二分查找（*binary search*）比顺序查找好得多。下面分析二分查找的原理。首先，把待查找的项称为目标项，而且假设列表中的各项按字母排序。然后，比较列表的中间项和目标项。如果两者相等，查找结束；假设目标项在列表中，如果中间项排在目标项前面，则目标项一定在后半部分项中；如果中间项在目标项后面，则目标项一定在前半部分项中。无论哪种情况，两项比较的结果都确定了下次查找的范围只有列表的一半。接着，继续使用这种方法，把需要查找的剩下一半的中间项与目标项比较。同样，这种方法会确定下一次查找的范围是当前查找范围的一半。以此类推，直到找到目标项或最终发现列表中没有目标项（见图 17.11）。这种方法非常有效率。假如有 127 个项，顺序查找平均要进行 64 次比较才能找到目标项或发现不在其中。但是二分查找最多只用进行 7 次比较。第 1 次比较剩下 63 项进行比较，第 2 次比较剩下 31 项进行比较，以此类推，第 6 次剩下最后 1 项进行比较，第 7 次比较确定剩下的这个项是否是目标项。一般而言，n 次比较能处理有 2^n-1 个元素的数组。所以项数越多，越能体现二分查找的优势。

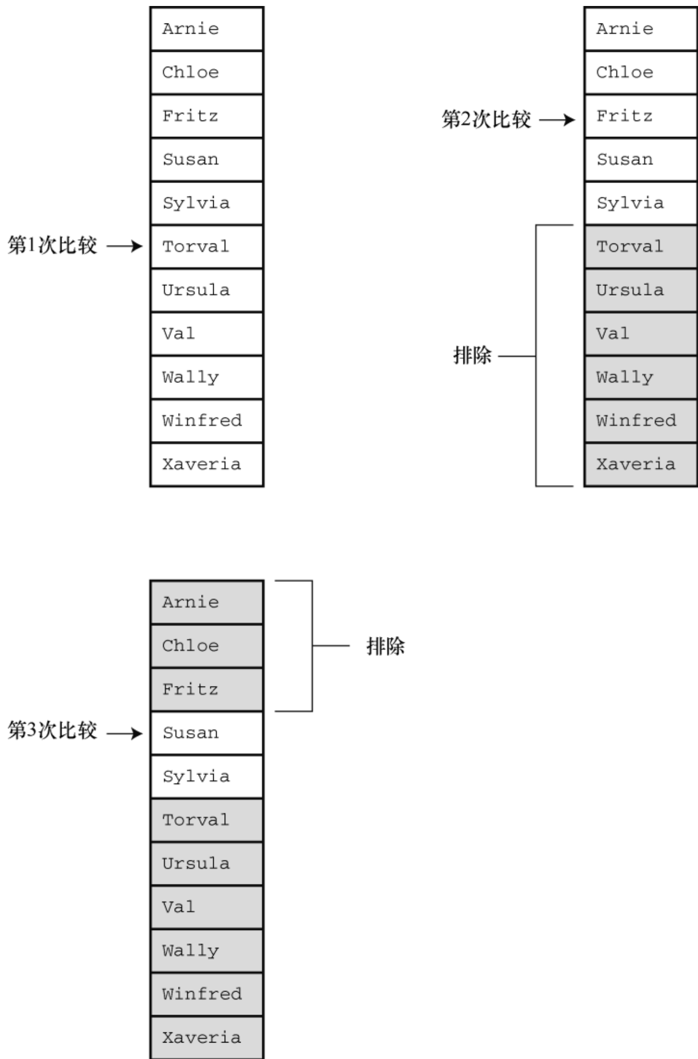


图 17.11 用二分查找法查找 Susan

用数组实现二分查找很简单，因为可以使用数组下标确定数组中任意部分的中点。只要把数组的首元素和尾元素的索引相加，得到的和再除以 2 即可。例如，内含 100 个元素的数组，首元素下标是 0，尾元素下标是 99，那么用于首次比较的中间项的下标应为(0+99)/2，得 49（整数除法）。如果比较的结果是下标为 49 的元素在目标项的后面，那么目标项的下标应在 0～48 的范围内。所以，第 2 次比较的中间项的下标应为(0+48)/2，得 24。如果中间项与目标项的比较结果是，中间项在目标项前面，那么第 3 次比较的中间项下标应为(25+48)/2，得 36。这体现了随机访问的特性，可以从一个位置跳至另一个位置，不用一次访问两位置之间的项。但是，链表只支持顺序访问，不提供跳至中间节点的方法。所以在链表中不能使用二分查找。

如前所述，选择何种数据类型取决于具体的问题。如果因频繁地插入和删除项导致经常调整大小，而且不需要经常查找，选择链表会更好。如果只是偶尔插入或删除项，但是经常进行查找，使用数组会更好。

如果需要一种既支持频繁插入和删除项又支持频繁查找的数据形式，数组和链表都无法胜任，怎么办？这种情况下应该选择二叉查找树。

17.7　二叉查找树

二叉查找树是一种结合了二分查找策略的链接结构。二叉树的每个节点都包含一个项和两个指向其他节点（称为子节点）的指针。图 17.12 演示了二叉查找树中的节点是如何链接的。二叉树中的每个节点都包含两个子节点——左节点和右节点，其顺序按照如下规定确定：左节点的项在父节点的项前面，右节点的项在父节点的项后面。这种关系存在于每个有子节点的节点中。进一步而言，所有可以追溯其祖先回到一个父节点的左节点的项，都在该父节点项的前面；所有以一个父节点的右节点为祖先的项，都在该父节点项的后面。图 17.12 中的树以这种方式储存单词。有趣的是，与植物学的树相反，该树的顶部被称为根（*root*）。树具有分层组织，所以以这种方式储存的数据也以等级或层次组织。一般而言，每级都有上一级和下一级。如果二叉树是满的，那么每一级的节点数都是上一级节点数的两倍。

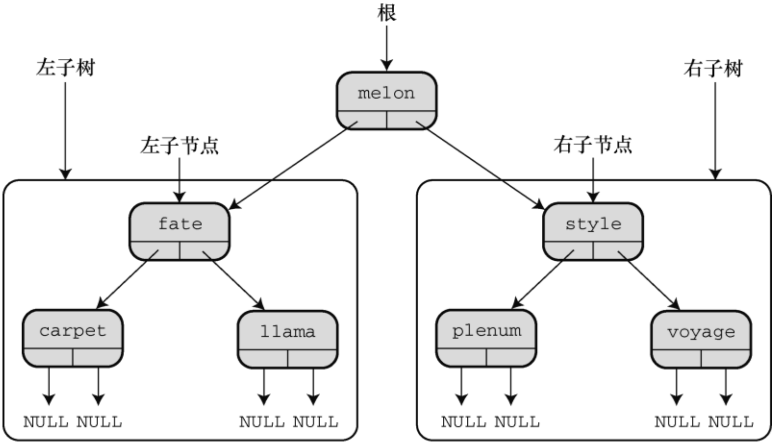


图 17.12　一个从存储单词的二叉树

二叉查找树中的每个节点是其后代节点的根，该节点与其后代节点构成称了一个子树（*subtree*）。如图 17.12 所示，包含单词 `fate`、`carpet` 和 `llama` 的节点构成了整个二叉树的左子树，而单词 `voyage` 是 `style-plenum-voyage` 子树的右子树。

假设要在二叉树中查找一个项（即目标项）。如果目标项在根节点项的前面，则只需查找左子树；如果目标项在根节点项的后面，则只需查找右子树。因此，每次比较就排除半个树。假设查找左子树，这意味着目标项与左子节点项比较。如果目标项在左子节点项的前面，则只需查找其后代节点的左半部分，以此类推。与二分查找类似，每次比较都能排除一半的可能匹配项。

我们用这种方法来查找 `puppy` 是否在图 17.12 的二叉树中。比较 `puppy` 和 `melon`（根节点项），如果 `puppy` 在该树中，一定在右子树中。因此，在右子树中比较 `puppy` 和 `style`，发现 `puppy` 在 `style` 前面，所以必须链接到其左节点。然后发现该节点是 `plenum`，在 `puppy` 前面。现在要向下链接到该节点的右子节点，但是没有右子节点了。所以经过 3 次比较后发现 `puppy` 不在该树中。

二叉查找树在链式结构中结合了二分查找的效率。但是，这样编程的代价是构建一个二叉树比创建一个链表更复杂。下面我们在下一个 ADT 项目中创建一个二叉树。

17.7.1　二叉树 ADT

和前面一样，先从概括地定义二叉树开始。该定义假设树不包含相同的项。许多操作与链表相同，区

别在于数据层次的安排。下面建立一个非正式的树定义：

类型名： 二叉查找树

类型属性： 二叉树要么是空节点的集合（空树），要么是有一个根节点的节点集合

每个节点都有两个子树，叫做左子树和右子树

每个子树本身也是一个二叉树，也有可能是空树

二叉查找树是一个有序的二叉树，每个节点包含一个项，

左子树的所有项都在根节点项的前面，右子树的所有项都在根节点项的后面

类型操作： 初始化树为空

确定树是否为空

确定树是否已满

确定树中的项数

在树中添加一个项

在树中删除一个项

在树中查找一个项

在树中访问一个项

清空树

17.7.2 二叉查找树接口

原则上，可以用多种方法实现二叉查找树，甚至可以通过操控数组下标用数组来实现。但是，实现二叉查找树最直接的方法是通过指针动态分配链式节点。因此我们这样定义：

```
typedef SOMETHING Item;

typedef struct trnode
{
    Item item;
    struct trnode * left;
    struct trnode * right;
} Trn;

typedef struct tree
{
    Trnode * root;
    int size;
} Tree;
```

每个节点包含一个项、一个指向左子节点的指针和一个指向右子节点的指针。可以把 `Tree` 定义为指向 `Trnode` 的指针类型，因为只需要知道根节点的位置就可访问整个树。然而，使用有成员大小的结构能很方便地记录树的大小。

我们要开发一个维护 `Nerfville` 宠物俱乐部的花名册，每一项都包含宠物名和宠物的种类。程序清单 17.10 就是该花名册的接口。我们把树的大小限制为 `10`，较小的树便于在树已满时测试程序的行为是否正确。当然，你也可以把 `MAXITEMS` 设置为更大的值。

程序清单 17.10　tree.h 接口头文件

```
/* tree.h -- 二叉查找数           */
/*             树种不允许有重复的项 */
#ifndef _TREE_H_
#define _TREE_H_
#include <stdbool.h>

/* 根据具体情况重新定义 Item */
#define SLEN 20
typedef struct item
{
    char petname[SLEN];
    char petkind[SLEN];
} Item;

#define MAXITEMS 10

typedef struct trnode
{
    Item item;
    struct trnode * left;      /* 指向左分支的指针  */
    struct trnode * right; /* 指向右分支的指针  */
} Trnode;

typedef struct tree
{
    Trnode * root;/* 指向根节点的指针          */
    int size;     /* 树的项数                  */
} Tree;

/* 函数原型 */

/* 操作：       把树初始化为空*/
/* 前提条件：   ptree 指向一个树     */
/* 后置条件：   树被初始化为空    */
void InitializeTree(Tree * ptree);

/* 操作：       确定树是否为空                          */
/* 前提条件：   ptree 指向一个树                        */
/* 后置条件：   如果树为空，该函数返回 true             */
/*                  否则，返回 false                    */
bool TreeIsEmpty(const Tree * ptree);

/* 操作：       确定树是否已满                          */
/* 前提条件：   ptree 指向一个树                        */
/* 后置条件：   如果树已满，该函数返回 true             */
/*                  否则，返回 false                    */
bool TreeIsFull(const Tree * ptree);

/* 操作：       确定树的项数                            */
/* 前提条件：   ptree 指向一个树                        */
```

```
/* 后置条件：    返回树的项数                          */
int TreeItemCount(const Tree * ptree);

/* 操作：        在树中添加一个项                      */
/* 前提条件：    pi 是待添加项的地址                   */
/*                   ptree 指向一个已初始化的树        */
/* 后置条件：    如果可以添加，该函数将在树中添加一个项  */
/*                   并返回 true；否则，返回 false     */
bool AddItem(const Item * pi, Tree * ptree);

/* 操作：        在树中查找一个项                      */
/* 前提条件：    pi 指向一个项                         */
/*                   ptree 指向一个已初始化的树        */
/* 后置条件：    如果在树中添加一个项，该函数返回 true   */
/*                   否则，返回 false                  */
bool InTree(const Item * pi, const Tree * ptree);

/* 操作：        从树中删除一个项                      */
/* 前提条件：    pi 是删除项的地址                     */
/*                   ptree 指向一个已初始化的树        */
/* 后置条件：    如果从树中成功删除一个项，该函数返回 true */
/*                   否则，返回 false                  */
bool DeleteItem(const Item * pi, Tree * ptree);

/* 操作：        把函数应用于树中的每一项              */
/* 前提条件：    ptree 指向一个树                      */
/*                   pfun 指向一个函数，               */
/*                   该函数接受一个 Item 类型的参数，并无返回值*/
/* 后置条件：    pfun 指向的这个函数为树中的每一项执行一次 */
void Traverse(const Tree * ptree, void(*pfun)(Item item));

/* 操作：        删除树中的所有内容                    */
/* 前提条件：    ptree 指向一个已初始化的树            */
/* 后置条件：    树为空                                */
void DeleteAll(Tree * ptree);

#endif
```

17.7.3 二叉树的实现

接下来，我们要实现 tree.h 中的每个函数。InitializeTree()、EmptyTree()、FullTree()和 TreeItems()函数都很简单，与链表 ADT、队列 ADT 类似，所以下面着重讲解其他函数。

1. 添加项

在树中添加一个项，首先要检查该树是否有空间放得下一个项。由于我们定义二叉树时规定其中的项不能重复，所以接下来要检查树中是否有该项。通过这两步检查后，便可创建一个新节点，把待添加项拷贝到该节点中，并设置节点的左指针和右指针都为 NULL。这表明该节点没有子节点。然后，更新 Tree 结

构的 size 成员，统计新增了一项。接下来，必须找出应该把这个新节点放在树中的哪个位置。如果树为空，则应设置根节点指针指向该新节点。否则，遍历树找到合适的位置放置该节点。AddItem()函数就根据这个思路来实现，并把一些工作交给几个尚未定义的函数：SeekItem()、MakeNode()和 AddNode()。

```
bool AddItem(const Item * pi, Tree * ptree)
{
    Trnode * new_node;
    if (TreeIsFull(ptree))
    {
        fprintf(stderr, "Tree is full\n");
        return false;           /* 提前返回    */
    }
    if (SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Attempted to add duplicate item\n");
        return false;           /* 提前返回    */
    }
    new_node = MakeNode(pi);    /* 指向新节点   */
    if (new_node == NULL)
    {
        fprintf(stderr, "Couldn't create node\n");
        return false;           /* 提前返回    */
    }
    /* 成功创建了一个新节点 */
    ptree->size++;
    if (ptree->root == NULL)             /* 情况 1：树为空         */
        ptree->root = new_node;          /* 新节点是根节点         */
    else                                 /* 情况 2：树不为空       */
        AddNode(new_node, ptree->root);/* 在树中添加一个节点 */
    return true;  /* 成功返回 */
}
```

SeekItem()、MakeNode()和 AddNode()函数不是 Tree 类型公共接口的一部分。它们是隐藏在 tree.c 文件中的静态函数，处理实现的细节（如节点、指针和结构），不属于公共接口。

MakeNode()函数相当简单，它处理动态内存分配和初始化节点。该函数的参数是指向新项的指针，其返回值是指向新节点的指针。如果 malloc()无法分配所需的内存，则返回空指针。只有成功分配了内存，MakeNode()函数才会初始化新节点。下面是 MakeNode()的代码：

```
static Trnode * MakeNode(const Item * pi)
{
    Trnode * new_node;
    new_node = (Trnode *) malloc(sizeof(Trnode));
    if (new_node != NULL)
    {
        new_node->item = *pi;
        new_node->left = NULL;
        new_node->right = NULL;
    }
    return new_node;
}
```

AddNode()函数是二叉查找树包中最麻烦的第 2 个函数。它必须确定新节点的位置，然后添加新节点。具体来说，该函数要比较新项和根项，以确定应该把新项放在左子树还是右子树中。如果新项是一个数字，

则使用<和>进行比较；如果新项是一个字符串，则使用 strcmp()函数来比较。但是，该项是内含两个字符串的结构，所以，必须自定义用于比较的函数。如果新项应放在左子树中，ToLeft()函数（稍后定义）返回 true；如果新项应放在右子树中，ToRight()函数（稍后定义）返回 true。这两个函数分别相当于<和>。假设把新项放在左子树中。如果左子树为空，AddNode()函数只需让左子节点指针指向新项即可。如果左子树不为空怎么办？此时，AddNode()函数应该把新项和左子节点中的项做比较，以确定新项应该放在该子节点的左子树还是右子树。这个过程一直持续到函数发现一个空子树为止，并在此此处添加新节点。递归是一种实现这种查找过程的方法，即把 AddNode()函数应用于子节点，而不是根节点。当左子树或右子树为空时，即当 root->left 或 root->right 为 NULL 时，函数的递归调用序列结束。记住，root 是指向当前子树顶部的指针，所以每次递归调用它都指向一个新的下一级子树（递归详见第 9 章）。

```
static void AddNode(Trnode * new_node, Trnode * root)
{
    if (ToLeft(&new_node->item, &root->item))
    {
        if (root->left == NULL)             /* 空子树 */
            root->left = new_node;          /* 所以，在此处添加节点 */
        else
            AddNode(new_node, root->left);  /* 否则，处理该子树*/
    }
    else if (ToRight(&new_node->item, &root->item))
    {
        if (root->right == NULL)
            root->right = new_node;
        else
            AddNode(new_node, root->right);
    }
    else                                    /* 不应含有重复的项 */
    {
        fprintf(stderr, "location error in AddNode()\n");
        exit(1);
    }
}
```

ToLeft()和 ToRight()函数依赖于 Item 类型的性质。Nerfville 宠物俱乐部的成员名按字母排序。如果两个宠物名相同，按其种类排序。如果种类也相同，这两项属于重复项，根据该二叉树的定义，这是不允许的。回忆一下，如果标准 C 库函数 strcmp()中的第 1 个参数表示的字符串在第 2 个参数表示的字符串前面，该函数则返回负数；如果两个字符串相同，该函数则返回 0；如果第 1 个字符串在第 2 个字符串后面，该函数则返回正数。ToRight()函数的实现代码与该函数类似。通过这两个函数完成比较，而不是直接在 AddNode()函数中直接比较，这样的代码更容易适应新的要求。当需要比较不同的数据形式时，就不必重写整个 AddNode()函数，只需重写 Toleft()和 ToRight()即可。

```
static bool ToLeft(const Item * i1, const Item * i2)
{
    int comp1;
    if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
        return true;
    else if (comp1 == 0 &&
        strcmp(i1->petkind, i2->petkind) < 0)
        return true;
    else
        return false;
}
```

2. 查找项

3 个接口函数都要在树中查找特定项：AddItem()、InItem()和 DeleteItem()。这些函数的实现中使用 SeekItem()函数进行查找。DeleteItem()函数有一个额外的要求：该函数要知道待删除项的父节点，以便在删除子节点后更新父节点指向子节点的指针。因此，我们设计 SeekItem()函数返回的结构包含两个指针：一个指针指向包含项的节点（如果未找到指定项则为 NULL）；一个指针指向父节点（如果该节点为根节点，即没有父节点，则为 NULL）。这个结构类型的定义如下：

```
typedef struct pair {
    Trnode * parent;
    Trnode * child;
} Pair;
```

SeekItem()函数可以用递归的方式实现。但是，为了给读者介绍更多编程技巧，我们这次使用 while 循环处理树中从上到下的查找。和 AddNode()一样，SeekItem()也使用 ToLeft()和 ToRight()在树中导航。开始时，SeekItem()设置 look.child 指针指向该树的根节点，然后沿着目标项应在的路径重置 look.child 指向后续的子树。同时，设置 look.parent 指向后续的父节点。如果没有找到匹配的项，look.child 则被设置为 NULL。如果在根节点找到匹配的项，则设置 look.parent 为 NULL，因为根节点没有父节点。下面是 SeekItem()函数的实现代码：

```
static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;
    if (look.child == NULL)
        return look;  /* 提前退出 */
    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else          /* 如果前两种情况都不满足，则必定是相等的情况 */
            break;        /* look.child 目标项的节点 */
    }
    return look;      /* 成功返回 */
}
```

注意，如果 SeekItem()函数返回一个结构，那么该函数可以与结构成员运算符一起使用。例如，AddItem()函数中有如下的代码：

```
if (SeekItem(pi, ptree).child != NULL)
```

有了 SeekItem()函数后，编写 InTree()公共接口函数就很简单了：

```
bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

3. 考虑删除项

删除项是最复杂的任务，因为必须重新连接剩余的子树形成有效的树。在准备编写这部分代码之前，必须明确需要做什么。

图 17.13 演示了最简单的情况。待删除的节点没有子节点，这样的节点被称为叶节点（*leaf*）。这种情况只需把父节点中的指针重置为 `NULL`，并使用 `free()` 函数释放已删除节点所占用的内存。

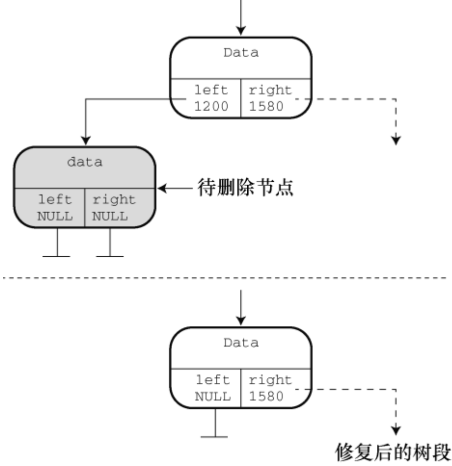


图 17.13 删除一个叶节点

删除带有一个子节点的情况比较复杂。删除该节点会导致其子树与其他部分分离。为了修正这种情况，要把被删除节点父节点中储存该节点的地址更新为该节点子树的地址（见图 17.14）。

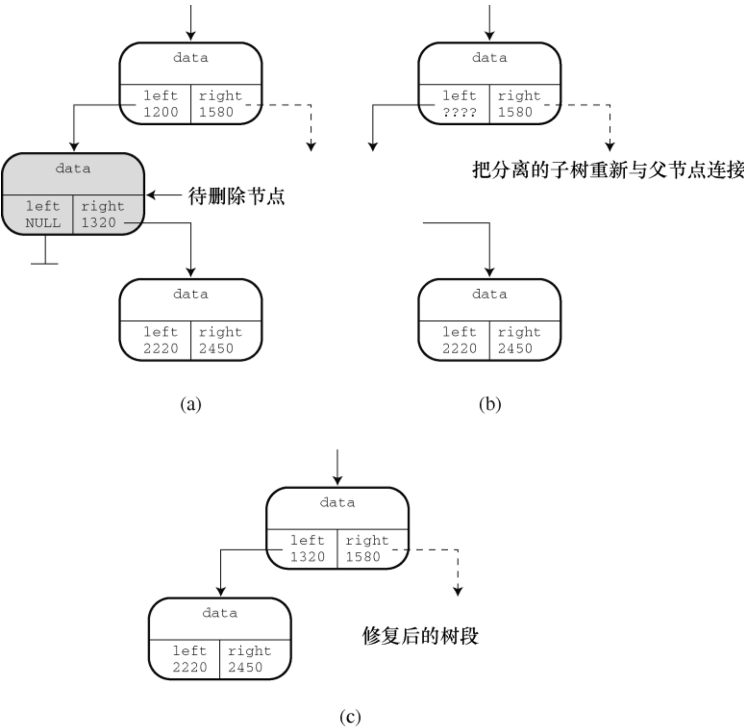


图 17.14 删除有一个子节点的节点

最后一种情况是删除有两个子树的节点。其中一个子树（如左子树）可连接在被删除节点之前连接的位置。但是，另一个子树怎么处理？牢记树的基本设计：左子树的所有项都在父节点项的前面，右子树的所有项都在父节点项的后面。也就是说，右子树的所有项都在左子树所有项的后面。而且，因为该右子树曾经是被删除节点的父节点的左子树的一部分，所以该右节点中的所有项在被删除节点的父节点项的前面。想像一下如何在树中从上到下查找该右子树的头所在的位置。它应该在被删除节点的父节点的前面，所以要沿着父节点的左子树向下找。但是，该右子树的所有项又在被删除节点左子树所有项的后面。因此要查看左子树的右支是否有新节点的空位。如果没有，就要沿着左子树的右支向下找，一直找到一个空位为止。图 17.15 演示了这种方法。

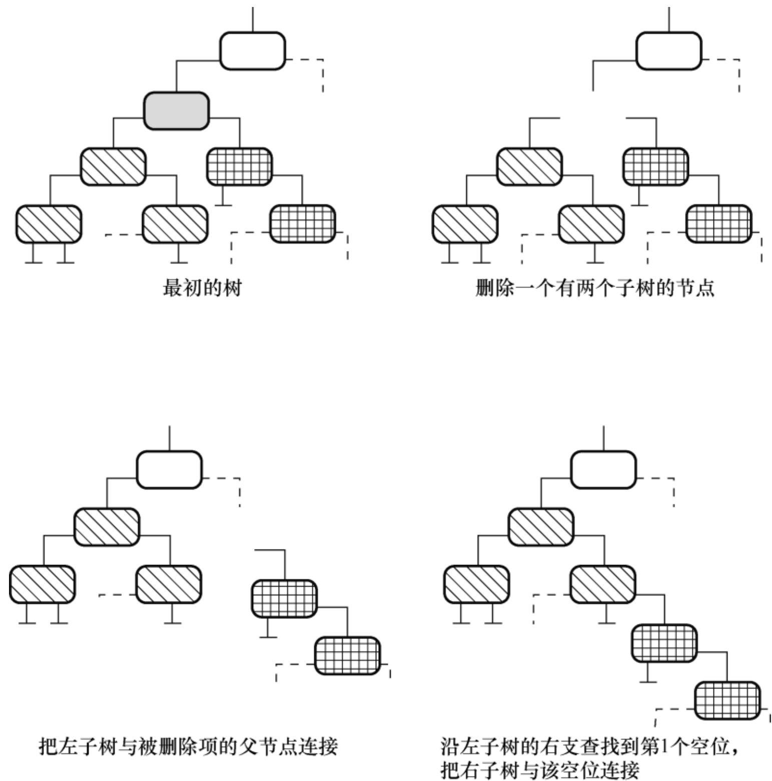


图 17.15　删除一个有两个子节点的项

① **删除一个节点**

现在可以设计所需的函数了，可以分成两个任务：第一个任务是把特定项与待删除节点关联；第二个任务是删除节点。无论哪种情况都必须修改待删除项父节点的指针。因此，要注意以下两点。

- 该程序必须标识待删除节点的父节点。
- 为了修改指针，代码必须把该指针的地址传递给执行删除任务的函数。

第一点稍后讨论，下面先分析第二点。要修改的指针本身是 `Trnode *`类型，即指向 `Trnode` 的指针。由于该函数的参数是该指针的地址，所以参数的类型是 `Trnode **`，即指向指针（该指针指向 `Trnode`）的指针。假设有合适的地址可用，可以这样编写执行删除任务的函数：

```
static void DeleteNode(Trnode **ptr)
/* ptr 是指向目标节点的父节点指针成员的地址 */
{
```

```
        Trnode * temp;
        if ((*ptr)->left == NULL)
        {
            temp = *ptr;
            *ptr = (*ptr)->right;
            free(temp);
        }
        else if ((*ptr)->right == NULL)
        {
            temp = *ptr;
            *ptr = (*ptr)->left;
            free(temp);
        }
        else /* 被删除的节点有两个子节点 */
        {
            /* 找到重新连接右子树的位置 */
            for (temp = (*ptr)->left; temp->right != NULL;
                temp = temp->right)
                continue;
            temp->right = (*ptr)->right;
            temp = *ptr;
            *ptr = (*ptr)->left;
            free(temp);
        }
    }
```

该函数显式处理了 3 种情况：没有左子节点的节点、没有右子节点的节点和有两个子节点的节点。无子节点的节点可作为无左子节点的节点的特例。如果该节点没有左子节点，程序就将右子节点的地址赋给其父节点的指针。如果该节点也没有右子节点，则该指针为 NULL。这就是无子节点情况的值。

注意，代码中用临时指针记录被删除节点的地址。被删除节点的父节点指针（*ptr）被重置后，程序会丢失被删除节点的地址，但是 free()函数需要这个信息。所以，程序把*ptr 的原始值储存在 temp 中，然后用 free()函数使用 temp 来释放被删除节点所占用的内存。

有两个子节点的情况，首先在 for 循环中通过 temp 指针从左子树的右半部分向下查找一个空位。找到空位后，把右子树连接于此。然后，再用 temp 保存被删除节点的位置。接下来，把左子树连接到被删除节点的父节点上，最后释放 temp 指向的节点。

注意，由于 ptr 的类型是 Trnode **，所以*ptr 的类型是 Trnode *，与 temp 的类型相同。

② 删除一个项

剩下的问题是把一个节点与特定项相关联。可以使用 SeekItem()函数来完成。回忆一下，该函数返回一个结构（内含两个指针，一个指针指向父节点，一个指针指向包含特定项的节点）。然后就可以通过父节点的指针获得相应的地址传递给 DeleteNode()函数。根据这个思路，DeleteNode()函数的定义如下：

```
    bool DeleteItem(const Item * pi, Tree * ptree)
    {
        Pair look;
        look = SeekItem(pi, ptree);
        if (look.child == NULL)
            return false;
        if (look.parent == NULL)    /* 删除根节点 */
            DeleteNode(&ptree->root);
```

```
        else if (look.parent->left == look.child)
            DeleteNode(&look.parent->left);
        else
            DeleteNode(&look.parent->right);
        ptree->size--;

        return true;
    }
```

首先，SeekItem()函数的返回值被赋给 look 类型的结构变量。如果 look.child 是 NULL，表明未找到指定项，DeleteItem()函数退出，并返回 false。如果找到了指定的 Item，该函数分 3 种情况来处理。第一种情况是，look.parent 的值为 NULL，这意味着该项在根节点中。在这情况下，不用更新父节点，但是要更新 Tree 结构中根节点的指针。因此，函数该函数把该指针的地址传递给 DeleteNode()函数。否则（即剩下两种情况），程序判断待删除节点是其父节点的左子节点还是右子节点，然后传递合适指针的地址。

注意，公共接口函数（DeleteItem()）处理的是最终用户所关心的问题（项和树），而隐藏的 DeleteNode()函数处理的是与指针相关的实质性任务。

4. 遍历树

遍历树比遍历链表更复杂，因为每个节点都有两个分支。这种分支特性很适合使用分而制之的递归（详见第 9 章）来处理。对于每一个节点，执行遍历任务的函数都要做如下的工作：

- 处理节点中的项；
- 处理左子树（递归调用）；
- 处理右子树（递归调用）。

可以把遍历分成两个函数来完成：Traverse()和 InOrder()。注意，InOrder()函数处理左子树，然后处理项，最后处理右子树。这种遍历树的顺序是按字母排序进行。如果你有时间，可以试试用不同的顺序，比如，项-左子树-右子树或者左子树-右子树-项，看看会发生什么。

```
void Traverse(const Tree * ptree, void(*pfun)(Item item))
{
    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}

static void InOrder(const Trnode * root, void(*pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}
```

5. 清空树

清空树基本上和遍历树的过程相同，即清空树的代码也要访问每个节点，而且要用 free()函数释放内存。除此之外，还要重置 Tree 类型结构的成员，表明该树为空。DeleteAll()函数负责处理 Tree 类型的结构，把释放内存的任务交给 DeleteAllNode()函数。DeleteAllNode()与 InOrder()函数的构造相同，它储存了指针的值 root->right，使其在释放根节点后仍然可用。下面是这两个函数的代码：

```
void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}
static void DeleteAllNodes(Trnode * root)
{
    Trnode * pright;
    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
        free(root);
        DeleteAllNodes(pright);
    }
}
```

6. 完整的包

程序清单 17.11 演示了整个 `tree.c` 的代码。`tree.h` 和 `tree.c` 共同组成了树的程序包。

程序清单 17.11 `tree.c` 程序

```
/* tree.c -- 树的支持函数 */
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "tree.h"

/* 局部数据类型 */
typedef struct pair {
    Trnode * parent;
    Trnode * child;
} Pair;

/* 局部函数的原型 */
static Trnode * MakeNode(const Item * pi);
static bool ToLeft(const Item * i1, const Item * i2);
static bool ToRight(const Item * i1, const Item * i2);
static void AddNode(Trnode * new_node, Trnode * root);
static void InOrder(const Trnode * root, void(*pfun)(Item item));
static Pair SeekItem(const Item * pi, const Tree * ptree);
static void DeleteNode(Trnode **ptr);
static void DeleteAllNodes(Trnode * ptr);

/* 函数定义 */
void InitializeTree(Tree * ptree)
{
    ptree->root = NULL;
    ptree->size = 0;
}

bool TreeIsEmpty(const Tree * ptree)
{
```

```
    if (ptree->root == NULL)
        return true;
    else
        return false;
}

bool TreeIsFull(const Tree * ptree)
{
    if (ptree->size == MAXITEMS)
        return true;
    else
        return false;
}

int TreeItemCount(const Tree * ptree)
{
    return ptree->size;
}

bool AddItem(const Item * pi, Tree * ptree)
{
    Trnode * new_node;

    if (TreeIsFull(ptree))
    {
        fprintf(stderr, "Tree is full\n");
        return false;          /* 提前返回    */
    }
    if (SeekItem(pi, ptree).child != NULL)
    {
        fprintf(stderr, "Attempted to add duplicate item\n");
        return false;          /* 提前返回    */
    }
    new_node = MakeNode(pi);   /* 指向新节点   */
    if (new_node == NULL)
    {
        fprintf(stderr, "Couldn't create node\n");
        return false;          /* 提前返回    */
    }
    /* 成功创建了一个新节点 */
    ptree->size++;

    if (ptree->root == NULL)            /* 情况 1：树为空        */
        ptree->root = new_node;         /* 新节点为树的根节点     */
    else                                /* 情况 2：树不为空       */
        AddNode(new_node, ptree->root);/* 在树中添加新节点       */

    return true;                        /* 成功返回             */
}

bool InTree(const Item * pi, const Tree * ptree)
{
    return (SeekItem(pi, ptree).child == NULL) ? false : true;
}
```

```
bool DeleteItem(const Item * pi, Tree * ptree)
{
    Pair look;

    look = SeekItem(pi, ptree);
    if (look.child == NULL)
        return false;

    if (look.parent == NULL)            /* 删除根节点项          */
        DeleteNode(&ptree->root);
    else if (look.parent->left == look.child)
        DeleteNode(&look.parent->left);
    else
        DeleteNode(&look.parent->right);
    ptree->size--;

    return true;
}

void Traverse(const Tree * ptree, void(*pfun)(Item item))
{

    if (ptree != NULL)
        InOrder(ptree->root, pfun);
}

void DeleteAll(Tree * ptree)
{
    if (ptree != NULL)
        DeleteAllNodes(ptree->root);
    ptree->root = NULL;
    ptree->size = 0;
}


/* 局部函数 */
static void InOrder(const Trnode * root, void(*pfun)(Item item))
{
    if (root != NULL)
    {
        InOrder(root->left, pfun);
        (*pfun)(root->item);
        InOrder(root->right, pfun);
    }
}

static void DeleteAllNodes(Trnode * root)
{
    Trnode * pright;

    if (root != NULL)
    {
        pright = root->right;
        DeleteAllNodes(root->left);
```

```
            free(root);
            DeleteAllNodes(pright);
        }
    }

    static void AddNode(Trnode * new_node, Trnode * root)
    {
        if (ToLeft(&new_node->item, &root->item))
        {
            if (root->left == NULL)              /* 空子树                 */
                root->left = new_node;           /* 把节点添加到此处       */
            else
                AddNode(new_node, root->left);   /* 否则处理该子树         */
        }
        else if (ToRight(&new_node->item, &root->item))
        {
            if (root->right == NULL)
                root->right = new_node;
            else
                AddNode(new_node, root->right);
        }
        else                                     /* 不允许有重复项         */
        {
            fprintf(stderr, "location error in AddNode()\n");
            exit(1);
        }
    }

    static bool ToLeft(const Item * i1, const Item * i2)
    {
        int comp1;

        if ((comp1 = strcmp(i1->petname, i2->petname)) < 0)
            return true;
        else if (comp1 == 0 &&strcmp(i1->petkind, i2->petkind) < 0)
            return true;
        else
            return false;
    }

    static bool ToRight(const Item * i1, const Item * i2)
    {
        int comp1;

        if ((comp1 = strcmp(i1->petname, i2->petname)) > 0)
            return true;
        else if (comp1 == 0 &&
            strcmp(i1->petkind, i2->petkind) > 0)
            return true;
        else
            return false;
    }

    static Trnode * MakeNode(const Item * pi)
    {
```

```
    Trnode * new_node;

    new_node = (Trnode *) malloc(sizeof(Trnode));
    if (new_node != NULL)
    {
        new_node->item = *pi;
        new_node->left = NULL;
        new_node->right = NULL;
    }

    return new_node;
}

static Pair SeekItem(const Item * pi, const Tree * ptree)
{
    Pair look;
    look.parent = NULL;
    look.child = ptree->root;

    if (look.child == NULL)
        return look;                    /* 提前返回    */

    while (look.child != NULL)
    {
        if (ToLeft(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->left;
        }
        else if (ToRight(pi, &(look.child->item)))
        {
            look.parent = look.child;
            look.child = look.child->right;
        }
        else                /* 如果前两种情况都不满足，则必定是相等的情况       */
            break;          /* look.child 目标项的节点                          */
    }

    return look;                    /* 成功返回 */
}

static void DeleteNode(Trnode **ptr)
/* ptr 是指向目标节点的父节点指针成员的地址 */
{
    Trnode * temp;

    if ( (*ptr)->left == NULL)
    {
        temp = *ptr;
        *ptr = (*ptr)->right;
        free(temp);
    }
    else if ( (*ptr)->right == NULL)
    {
```

```
            temp = *ptr;
            *ptr = (*ptr)->left;
            free(temp);
        }
        else    /* 被删除的节点有两个子节点 */
        {
            /* 找到重新连接右子树的位置 */
            for (temp = (*ptr)->left; temp->right != NULL;temp = temp->right)
                continue;
            temp->right = (*ptr)->right;
            temp = *ptr;
            *ptr = (*ptr)->left;
            free(temp);
        }
    }
```

17.7.4　使用二叉树

现在，有了接口和函数的实现，就可以使用它们了。程序清单 17.12 中的程序以菜单的方式提供选择：向俱乐部成员花名册添加宠物、显示成员列表、报告成员数量、核实成员及退出。main()函数很简单，主要提供程序的大纲。具体工作主要由支持函数来完成。

程序清单 17.12 petclub.c **程序**

```
/* petclub.c -- 使用二叉查找数 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "tree.h"

char menu(void);
void addpet(Tree * pt);
void droppet(Tree * pt);
void showpets(const Tree * pt);
void findpet(const Tree * pt);
void printitem(Item item);
void uppercase(char * str);
char * s_gets(char * st, int n);

int main(void)
{
    Tree pets;
    char choice;

    InitializeTree(&pets);
    while ((choice = menu()) != 'q')
    {
        switch (choice)
        {
        case 'a':  addpet(&pets);
            break;
        case 'l':  showpets(&pets);
            break;
```

```
        case 'f':  findpet(&pets);
            break;
        case 'n':  printf("%d pets in club\n",
            TreeItemCount(&pets));
            break;
        case 'd':  droppet(&pets);
            break;
        default:  puts("Switching error");
        }
    }
    DeleteAll(&pets);
    puts("Bye.");

    return 0;
}

char menu(void)
{
    int ch;

    puts("Nerfville Pet Club Membership Program");
    puts("Enter the letter corresponding to your choice:");
    puts("a) add a pet          l) show list of pets");
    puts("n) number of pets     f) find pets");
    puts("d) delete a pet       q) quit");
    while ((ch = getchar()) != EOF)
    {
        while (getchar() != '\n')  /* 处理输入行的剩余内容 */
            continue;
        ch = tolower(ch);
        if (strchr("alrfndq", ch) == NULL)
            puts("Please enter an a, l, f, n, d, or q:");
        else
            break;
    }
    if (ch == EOF)       /* 使程序退出 */
        ch = 'q';

    return ch;
}

void addpet(Tree * pt)
{
    Item temp;

    if (TreeIsFull(pt))
        puts("No room in the club!");
    else
    {
        puts("Please enter name of pet:");
        s_gets(temp.petname, SLEN);
        puts("Please enter pet kind:");
        s_gets(temp.petkind, SLEN);
        uppercase(temp.petname);
        uppercase(temp.petkind);
        AddItem(&temp, pt);
```

```
        }
    }

    void showpets(const Tree * pt)
    {
        if (TreeIsEmpty(pt))
            puts("No entries!");
        else
            Traverse(pt, printitem);
    }

    void printitem(Item item)
    {
        printf("Pet: %-19s  Kind: %-19s\n", item.petname,item.petkind);
    }

    void findpet(const Tree * pt)
    {
        Item temp;

        if (TreeIsEmpty(pt))
        {
            puts("No entries!");
            return;     /* 如果树为空，则退出该函数 */
        }

        puts("Please enter name of pet you wish to find:");
        s_gets(temp.petname, SLEN);
        puts("Please enter pet kind:");
        s_gets(temp.petkind, SLEN);
        uppercase(temp.petname);
        uppercase(temp.petkind);
        printf("%s the %s ", temp.petname, temp.petkind);
        if (InTree(&temp, pt))
            printf("is a member.\n");
        else
            printf("is not a member.\n");
    }

    void droppet(Tree * pt)
    {
        Item temp;

        if (TreeIsEmpty(pt))
        {
            puts("No entries!");
            return;     /* 如果树为空，则退出该函数 */
        }

        puts("Please enter name of pet you wish to delete:");
        s_gets(temp.petname, SLEN);
        puts("Please enter pet kind:");
        s_gets(temp.petkind, SLEN);
        uppercase(temp.petname);
        uppercase(temp.petkind);
```

```
    printf("%s the %s ", temp.petname, temp.petkind);
    if (DeleteItem(&temp, pt))
        printf("is dropped from the club.\n");
    else
        printf("is not a member.\n");
}

void uppercase(char * str)
{
    while (*str)
    {
        *str = toupper(*str);
        str++;
    }
}
char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');    // 查找换行符
        if (find)                 // 如果地址不是 NULL,
            *find = '\0';           // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;      // 处理输入行的剩余内容
    }
    return ret_val;
}
```

该程序把所有字母都转换为大写字母，所以 SNUFFY、Snuffy 和 snuffy 都被视为相同。下面是该程序的一个运行示例：

```
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet           l) show list of pets
n) number of pets      f) find pets
q) quit
a
Please enter name of pet:
Quincy
Please enter pet kind:
pig
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet           l) show list of pets
n) number of pets      f) find pets
q) quit
a
Please enter name of pet:
Bennie Haha
Please enter pet kind:
```

```
parrot
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet           l) show list of pets
n) number of pets      f) find pets
q) quit
a
Please enter name of pet:
Hiram Jinx
Please enter pet kind:
domestic cat
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet           l) show list of pets
n) number of pets      f) find pets
q) quit
n
3 pets in club
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet           l) show list of pets
n) number of pets      f) find pets
q) quit
l
Pet: BENNIE HAHA            Kind: PARROT
Pet: HIRAM JINX             Kind: DOMESTIC CAT
Pet: QUINCY                 Kind: PIG
Nerfville Pet Club Membership Program
Enter the letter corresponding to your choice:
a) add a pet           l) show list of pets
n) number of pets      f) find pets
q) quit
q
Bye.
```

17.7.5 树的思想

二叉查找树也有一些缺陷。例如，二叉查找树只有在满员（或平衡）时效率最高。假设要储存用户随机输入的单词。该树的外观应如图 17.12 所示。现在，假设用户按字母顺序输入数据，那么每个新节点应该被添加到右边，该树的外观应如图 17.16 所示。图 17.12 所示是平衡的树，图 17.16 所示是不平衡的树。查找这种树并不比查找链表要快。

避免串状树的方法之一是在创建树时多加注意。如果树或子树的一边或另一边太不平衡，就需要重新排列节点使之恢复平衡。与此类似，可能在进行删除操作后要重新排列树。俄国数学家 Adel'son-Vel'skii 和 Landis 发明了一种算法来解决这个问题。根据他们的算法创建的树称为 AVL 树。因为要重构，所以创建一个平衡的树所花费的时间更多，但是这样的树可以确保最大化搜索效率。

你可能需要一个能储存相同项的二叉查找树。例如，在分析一些文本时，统计某个单词在文本中出现的次数。一种方法是把 `Item` 定义成包含一个单词和一个数字的结构。第一次遇到一个单词时，将其添加到树中，并且该单词的数量加 `1`。下一次遇到同样的单词时，程序找到包含该单词的节点，并递增表示该单词数量的值。把基本二叉查找树修改成具有这一特性，不费多少工夫。

考虑 Nerfville 宠物俱乐部的示例，有另一种情况。示例中的树根据宠物的名字和种类进行排列，所以，可以把名为 Sam 的猫储存在一个节点中，把名为 Sam 的狗储存在另一节点中，把名为 Sam 的山羊储存在

第 3 个节点中。但是，不能储存两只名为 Sam 的猫。另一种方法是以名字来排序，但是这样做只能储存一个名为 Sam 的宠物。还需要要把 `Item` 定义成多个结构，而不是一个结构。第一次出现 Sally 时，程序创建一个新的节点，并创建一个新的列表，然后把 Sally 及其种类添加到列表中。下一次出现 Sally 时，程序将定位到之前储存 Sally 的节点，并把新的数据添加到结构列表中。

提示　插件库

读者可能意识到实现一个像链表或树这样的 ADT 比较困难，很容易犯错。插件库提供了一种可选的方法：让其他人来完成这些工作和测试。在学完本章这两个相对简单的例子后，读者应该能很好地理解和认识这样的库。

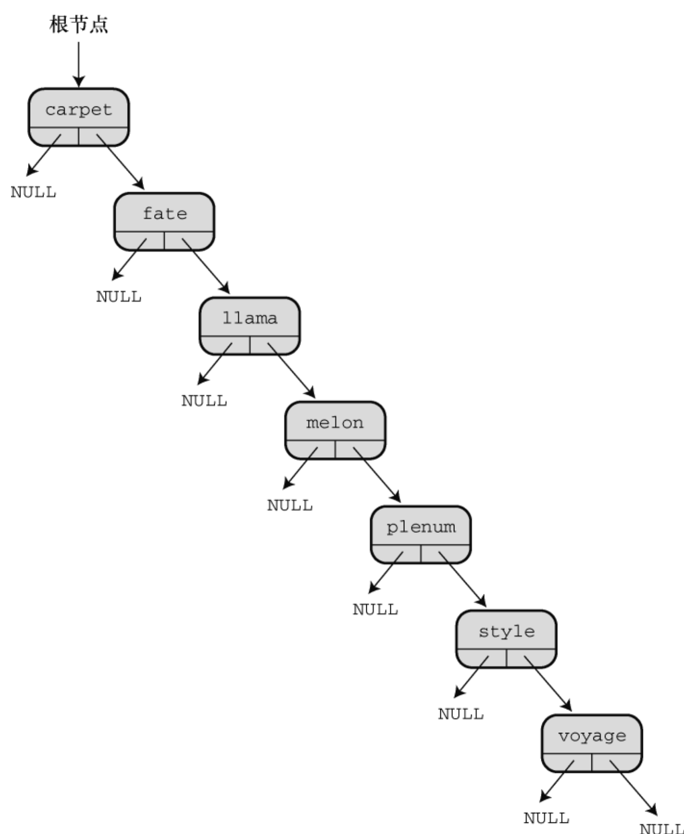


图 17.16　不平衡的二叉查找树

17.8　其他说明

本书中，我们涵盖了 C 语言的基本特性，但是只是简要介绍了库。ANSI C 库中包含多种有用的函数。绝大部分实现都针对特定的系统提供扩展库。基于 Windows 的编译器支持 Windows 图形接口。Macintosh C 编译器提供访问 Macintosh 工具箱的函数，以便编写具有标准 Macintosh 接口或 iOS 系统的程序产品，如 iPhone 或 iPad。与此类似，还有一些工具用于创建 Linux 程序的图形接口。花时间查看你的系统提供什么。如果没有你想要的工具，就自己编写函数。这是 C 的一部分。如果认为自己能编写一个更好的（如，输入函数），那就去做！随着你不断练习并提高自己的编程技术，会从一名新手成为经验丰富的资深程序员。

如果对链表、队列和树的相关概念感兴趣或觉得很有用，可以阅读其他相关的书籍，学习高级编程技巧。计算机科学家在开发和分析算法以及如何表示数据方面投入了大量的时间和精力。也许你会发现已经有人开发了你正需要的工具。

学会 C 语言后，你可能想研究 C++、Objectiv C 或 Java。这些都是以 C 为基础的面向对象（*object-oriented*）语言。C 已经涵盖了从简单的 `char` 类型变量到大型且复杂的结构在内的数据对象。面向对象语言更进一步发展了对象的观点。例如，对象的性质不仅包括它所储存的信息类型，而且还包括了对其进行的操作类型。本章介绍的 ADT 就遵循了这种模式。而且，对象可以继承其他对象的属性。OOP 提供比 C 更高级的抽象，很适合编写大型程序。

请参阅附录 B 中的参考资料 I“补充阅读”中找到你感兴趣的书籍。

17.9　关键概念

一种数据类型通过以下几点来表征：如何构建数据、如何储存数据、有哪些可能的操作。抽象数据类型（ADT）以抽象的方式指定构成某种类型特征的属性和操作。从概念上看，可以分两步把 ADT 翻译成一种特定的编程语言。第 1 步是定义编程接口。在 C 中，通过使用头文件定义类型名，并提供与允许的操作相应的函数原型来实现。第 2 步是实现接口。在 C 中，可以用源代码文件提供与函数原型相应的函数定义来实现。

17.10　本章小结

链表、队列和二叉树是 ADT 在计算机程序设计中常用的示例。通常用动态内存分配和链式结构来实现它们，但有时用数组来实现会更好。

当使用一种特定类型（如队列或树）进行编程时，要根据类型接口来编写程序。这样，在修改或改进实现时就不用更改使用接口的那些程序。

17.11　复习题

1. 定义一种数据类型涉及哪些内容？
2. 为什么程序清单 17.2 只能沿一个方向遍历链表？如何修改 `struct film` 定义才能沿两个方向遍历链表？
3. 什么是 ADT？
4. `QueueIsEmpty()` 函数接受一个指向 `queue` 结构的指针作为参数，但是也可以将其编写成接受一个 `queue` 结构作为参数。这两种方式各有什么优缺点？
5. 栈（*stack*）是链表系列的另一种数据形式。在栈中，只能在链表的一端添加和删除项，项被“压入”栈和“弹出”栈。因此，栈是一种 LIFO（即后进先出 *last in,first out*）结构。

   a．设计一个栈 ADT

   b．为栈设计一个 C 编程接口，例如 `stack.h` 头文件
6. 在一个含有 3 个项的分类列表中，判断一个特定项是否在该列表中，用顺序查找和二叉查找方法分别需要最多多少次？当列表中有 1023 个项时分别是多少次？65535 个项是分别是多少次？
7. 假设一个程序用本章介绍的算法构造了一个储存单词的二叉查找树。假设根据下面所列的顺序输入

单词，请画出每种情况的树：

a．nice food roam dodge gate office wave

b．wave roam office nice gate food dodge

c．food dodge roam wave office gate nice

d．nice roam office food wave gate dodge

8．考虑复习题 7 构造的二叉树，根据本章的算法，删除单词 food 之后，各树是什么样子？

17.12　编程练习

1．修改程序清单 17.2，让该程序既能正序也能逆序显示电影列表。一种方法是修改链表的定义，可以双向遍历链表。另一种方法是用递归。

2．假设 list.h（程序清单 17.3）使用下面的 list 定义：

```
typedef struct list
{
    Node * head;    /* 指向 list 的开头 */
    Node * end;/* 指向 list 的末尾 */
} List;
```

重写 list.c（程序清单 17.5）中的函数以适应新的定义，并通过 films.c（程序清单 17.4）测试最终的代码。

3．假设 list.h（程序清单 17.3）使用下面的 list 定义：

```
#define MAXSIZE 100
typedef struct list
{
    Item entries[MAXSIZE];  /* 内含项的数组 */
    int items;              /* list 中的项数 */
} List;
```

重写 list.c（程序清单 17.5）中的函数以适应新的定义，并通过 films.c（程序清单 17.4）测试最终的代码。

4．重写 mall.c（程序清单 17.7），用两个队列模拟两个摊位。

5．编写一个程序，提示用户输入一个字符串。然后该程序把该字符串的字符逐个压入一个栈（参见复习题 5），然后从栈中弹出这些字符，并显示它们。结果显示为该字符串的逆序。

6．编写一个函数接受 3 个参数：一个数组名（内含已排序的整数）、该数组的元素个数和待查找的整数。如果待查找的整数在数组中，那么该函数返回 1；如果该数不在数组中，该函数则返回 0。用二分查找法实现。

7．编写一个程序，打开和读取一个文本文件，并统计文件中每个单词出现的次数。用改进的二叉查找树储存单词及其出现的次数。程序在读入文件后，会提供一个有 3 个选项的菜单。第 1 个选项是列出所有的单词和出现的次数。第 2 个选项是让用户输入一个单词，程序报告该单词在文件中出现的次数。第 3 个选项是退出。

8．修改宠物俱乐部程序，把所有同名的宠物都储存在同一个节点中。当用户选择查找宠物时，程序应询问用户该宠物的名字，然后列出该名字的所有宠物（及其种类）。

附录 A

复习题答案

A.1　第 1 章复习题答案

1. 完美的可移植程序是，其源代码无需修改就能在不同计算机系统中成功编译的程序。
2. 源代码文件包含程序员使用的任何编程语言编写的代码。目标代码文件包含机器语言代码，它不必是完整的程序代码。可执行文件包含组成可执行程序的完整机器语言代码。
3. （1）定义程序目标；（2）设计程序；（3）编写程序；（4）编译程序；（5）运行程序；（6）测试和调试程序；（7）维护和修改程序。
4. 编译器把源代码（如，用 C 语言编写的代码）翻译成等价的机器语言代码（也叫作**目标代码**）。
5. 链接器把编译器翻译好的源代码以及库代码和启动代码组合起来，生成一个可执行程序。

A.2　第 2 章复习题答案

1. 它们都叫作函数。
2. 语法错误违反了组成语句或程序的规则。这是一个有语法错误的英文例子：Me speak English good.。这是一个有语法错误的 C 语言例子：`printf"Where are the parentheses?";`。
3. 语义错误是指含义错误。这是一个有语义错误的英文例子：This sentence isexcellent Czech.[1]。这是一个有语义错误的C语言例子： `thrice_n = 3 + n;`[2]。
4. 第 1 行：以一个#开始；`studio.h` 应改成 `stdio.h`；然后用一对尖括号把 `stdio.h` 括起来。

   第 2 行：把`{}`改成`()`；注释末尾把`/*`改成`*/`。

   第 3 行：把`(`改成`{`

   第 4 行：`int s` 末尾加上一个分号。

   第 5 行没问题。

   第 6 行：把`:=`改成，赋值用`=`，而不是用`:=`（这说明 Indiana Sloth 了解 Pascal）。另外，用于赋值的值 56 也不对，一年有 52 周，不是 56 周。

   第 7 行应该是：`printf("There are %d weeks in a year.\n", s);`

   第 9 行：原程序中没有第 9 行，应该在该行加上一个右花括号`}`。

   修改后的程序如下：

```
#include <stdio.h>
int main(void) /* this prints the number of weeks in a year */
{
```

[1] 这句英文翻译成中文是“这句话是出色的捷克人”。显然不知所云，这就是语言中的语义错误。——译者注

[2] `thrice_n` 本应表示 n 的 3 倍，但是 `3 + n` 表示的并不是 n 的 3 倍，应该用 `3*n` 来表示。——译者注

```
    int s;

    s = 52;
    printf("There are %d weeks in a year.\n", s);
    return 0;
}
```

5. a. Baa Baa Black Sheep.Have you any wool?（注意，Sheep.和 Have 之间没有空格）

   b. Begone!

   O creature of lard!

   c. What?

   No/nfish?

   （注意斜杠/和反斜杠\的效果不同，/只是一个普通的字符，原样打印）

   d. 2 + 2 = 4

   （注意，每个%d 与列表中的值相对应。还要注意，+的意思是加法，可以在 printf()语句内部计算）

6. 关键字是 int 和 char（main 是一个函数名；function 是函数的意思；=是一个运算符）。

7. printf("There were %d words and %d lines.\n", words, lines);

8. 执行完第 7 行后，a 是 5，b 是 2。执行完第 8 行后，a 和 b 都是 5。执行完第 9 行后，a 和 b 仍然是 5（注意，a 不会是 2，因为在执行 a = b;时，b 的值已经被改为 5）。

9. 执行完第 7 行后，x 是 10，b 是 5。执行完第 8 行后，x 是 10，y 是 15。执行完第 9 行后，x 是 150，y 是 15。

A.3　第 3 章复习题答案

1. a. int 类型，也可以是 short 类型或 unsigned short 类型。人口数是一个整数。

   b. float 类型，价格通常不是一个整数（也可以使用 double 类型，但实际上不需要那么高的精度）。

   c. char 类型。

   d. int 类型，也可以是 unsigned 类型。

2. 原因之一：在系统中要表示的数超过了 int 可表示的范围，这时要使用 long 类型。原因之二：如果要处理更大的值，那么使用一种在所有系统上都保证至少是 32 位的类型，可提高程序的可移植性。

3. 如果要正好获得 32 位的整数，可以使用 int32_t 类型。要获得可储存至少 32 位整数的最小类型，可以使用 int_least32_t 类型。如果要为 32 位整数提供最快的计算速度，可以选择 int_fast32_t 类型（假设你的系统已定义了上述类型）。

4. a. char 类型常量（但是储存为 int 类型）

   b. int 类型常量

   c. double 类型常量

   d. unsigned int 类型常量，十六进制格式

   e. double 类型常量

5. 第 1 行：应该是#include <stdio.h>

   第 2 行：应该是 int main(void)

第 3 行：把(改为{

第 4 行：g 和 h 之间的;改成,

第 5 行：没问题

第 6 行：没问题

第 7 行：虽然这数字比较大，但在 e 前面应至少有一个数字，如 1e21 或 1.0e21 都可以。

第 8 行：没问题，至少没有语法问题。

第 9 行：把)改成}

除此之外，还缺少一些内容。首先，没有给 rate 变量赋值；其次未使用 h 变量；而且程序不会报告计算结果。虽然这些错误不会影响程序的运行（编译器可能给出变量未被使用的警告），但是它们确实与程序设计的初衷不符合。另外，在该程序的末尾应该有一个 return 语句。

下面是一个正确的版本，仅供参考：

```
#include <stdio.h>
int main(void)
{
    float g, h;
    float tax, rate;

    rate = 0.08;
    g = 1.0e5;
    tax = rate*g;
    h = g + tax;
    printf("You owe $%f plus $%f in taxes for a total of $%f.\n", g, tax, h);
    return 0;
}
```

6.

常量	类型	转换说明（%转换字符）
12	int	%d
0X3	unsigned int	%#X
'C'	char（实际上是 int）	%c
2.34E07	double	%e
'\040'	char（实际上是 int）	%c
7.0	double	%f
6L	long	%ld
6.0f	float	%f
0x5.b6p12	float	%a

7.

常量	类型	转换说明（%转换字符）
012	unsigned int	%#o
2.9e05L	long double	%Le
's'	char（实际上是 int）	%c
100000	long	%ld
'\n'	char（实际上是 int）	%c
20.0f	float	%f
0x44	unsigned int	%x
-40	int	%d

8. printf("The odds against the %d were %ld to 1.\n", imate, shot);

```
printf("A score of %f is not an %c grade.\n", log, grade);
```

9.
```
ch = '\r';
ch = 13;
ch = '\015'
ch = '\xd'
```

10. 最前面缺少一行（第 0 行：#include <stdio.h>

    第 1 行：使用/*和*/把注释括起来，或者在注释前面使用//。

    第 3 行：int cows, legs;

    第 4 行：country？\n");

    第 5 行：把%c 改为%d，把 legs 改为&legs。

    第 7 行：把%f 改为%d。

    另外，在程序末尾还要加上 return 语句。

    下面是修改后的版本：

```
#include <stdio.h>
int main(void) /* this program is perfect */
{
    int cows, legs;
    printf("How many cow legs did you count?\n");
    scanf("%d", &legs);
    cows = legs / 4;
    printf("That implies there are %d cows.\n", cows);
    return 0;
}
```

11. a. 换行字符

    b. 反斜杠字符

    c. 双引号字符

    d. 制表字符

A.4　第 4 章复习题答案

1. 程序不能正常运行。第 1 个 scanf()语句只读取用户输入的名，而用户输入的姓仍留在输入缓冲区中（缓冲区是用于储存输入的临时存储区）。下一条 scang()语句在输入缓冲区查找重量时，从上次读入结束的地方开始读取。这样就把留在缓冲区的姓作为体重来读取，导致 scanf()读取失败。另一方面，如果在要求输入姓名时输入 Lasha 144，那么程序会把 144 作为用户的体重（虽然用户是在程序提示输入体重之前输入了 144）。

2. a. He sold the painting for $234.50.

   b. Hi!（注意，第 1 个字符是字符常量；第 2 个字符由十进制整数转换而来；第 3 个字符是八进制字符常量的 ASCII 表示）

   c. His Hamlet was funny without being vulgar.
      has 42 characters.

   d. Is 1.20e+003 the same as 1201.00?

3. 在这条语句中使用\"：`printf("\"%s\"\nhas %d characters.\n", Q, strlen(Q));`

4. 下面是修改后的程序：

```
#include <stdio.h>      /* 别忘了要包含合适的头文件 */
#define B "booboo"      /* 添加#、双引号 */
#define X 10            /* 添加# */
int main(void)          /* 不是main(int) */
{
    int age;
    int xp;             /* 声明所有的变量 */
    char name[40];      /* 把name声明为数组 */

    printf("Please enter your first name.\n"); /* 添加\n，提高可读性 */
    scanf("%s", name);
    printf("All right, %s, what's your age?\n", name); /* %s用于打印字符串*/
    scanf("%d", &age); /* 把%f改成%d，把age改成&age */
    xp = age + X;
    printf("That's a %s! You must be at least %d.\n", B, xp);
    return 0; /* 不是rerun */
}
```

5. 记住，要打印%必须用%%：

```
printf("This copy of \"%s\" sells for $%0.2f.\n", BOOK, cost);
printf("That is %0.0f%% of list.\n", percent);
```

6. a. `%d`

   b. `%4X`

   c. `%10.3f`

   d. `%12.2e`

   e. `%-30s`

7. a. `%15lu`

   b. `%#4x`

   c. `%-12.2E`

   d. `%+10.3f`

   e. `%8.8s`

8. a. `%6.4d`

   b. `%*o`

   c. `%2c`

   d. `%+0.2f`

   e. `%-7.5s`

9. a.
```
int dalmations;
scanf("%d", &dalmations);
```

b. `float kgs, share;`
```
scanf("%f%f", &kgs, &share);
```
（注意：对于本题的输入，可以使用转换字符 e、f 和 g。另外，除了%c 之外，在%和转换字符之间加空格不会影响最终的结果）

c. `char pasta[20];`
```
scanf("%s", pasta);
```
d. `char action[20];`
```
int value;
scanf("%s %d", action, &value);
```
e. `int value;`
```
scanf("%*s %d", &value);
```
10. 空白包括空格、制表符和换行符。C 语言使用空白分隔记号。scanf()使用空白分隔连续的输入项。

11. %z 中的 z 是修饰符，不是转换字符，所以要在修饰符后面加上一个它修饰的转换字符。可以使用%zd 打印十进制数，或用不同的说明符打印不同进制的数，例如，%zx 打印十六进制的数。

12. 可以分别把(和)替换成{和}。但是预处理器无法区分哪些圆括号应替换成花括号，哪些圆括号不能替换成花括号。因此，
```
#define ( {
#define ) }
int main(void)
(
    printf("Hello, O Great One!\n");
)
```
将变成：
```
int main{void}
{
    printf{"Hello, O Great One!\n"};
}
```

A.5　第 5 章复习题答案

1. a. 30

   b. 27（不是 3）。(12+6)/(2*3)得 3。

   c. x = 1，y = 1（整数除法）。

   d. x = 3（整数除法），y = 9。

2. a. 6（由 3 + 3.3 截断而来）

   b. 52

   c. 0（0 * 22.0 的结果）

   d. 13（66.0 / 5 或 13.2，然后把结果赋给 int 类型变量）

3. a. 37.5（7.5 * 5.0 的结果）

   b. 1.5（30.0 / 20.0 的结果）

c. 35（7 * 5 的结果）

d. 37（150 / 4 的结果）

e. 37.5（7.5 * 5 的结果）

f. 35.0（7 * 5.0 的结果）

4. 第 0 行：应增加一行#include <stdio.h>。

   第 3 行：末尾用分号，而不是逗号。

   第 6 行：while 语句创建了一个无限循环。因为 i 的值始终为 1，所以它总是小于 30。推测一下，应该是想写 while(i++ < 30)。

   第 6～8 行：这样的缩进布局不能使第 7 行和第 8 行组成一个代码块。由于没有用花括号括起来，while 循环只包括第 7 行，所以要添加花括号。

   第 7 行：因为 1 和 i 都是整数，所以当 i 为 1 时，除法的结果是 1；当 i 为更大的数时，除法结果为 0。用 n = 1.0/i，i 在除法运算之前会被转换为浮点数，这样就能得到非零值。

   第 8 行：在格式化字符串中没有换行符（\n），这导致数字被打印成一行。

   第 10 行：应该是 return 0;

   下面是正确的版本：

```
#include <stdio.h>
int main(void)
{
    int i = 1;
    float n;
    printf("Watch out! Here come a bunch of fractions!\n");
    while (i++ < 30)
    {
        n = 1.0/i;
        printf(" %f\n", n);
    }
    printf("That's all, folks!\n");
    return 0;
}
```

5. 这个版本最大的问题是测试条件（sec 是否大于 0？）和 scanf()语句获取 sec 变量的值之间的关系。具体地说，第一次测试时，程序尚未获得 sec 的值，用来与 0 作比较的是正好在 sec 变量内存位置上的一个垃圾值。一个比较笨拙的方法是初始化 sec（如，初始化为 1）。这样就可通过第一次测试。不过，还有另一个问题。当最后输入 0 结束程序时，在循环结束之前不会检查 sec，所以 0 也被打印了出来。因此，更好的方法是在 while 测试之前使用 scanf()语句。可以这样修改：

```
scanf("%d", &sec);
while ( sec > 0 ) {
    min = sec/S_TO_M;
    left = sec % S_TO_M;
    printf("%d sec is %d min, %d sec. \n", sec, min, left);
    printf("Next input?\n");
    scanf("%d", &sec);
}
```

   while 循环第一轮迭代使用的是 scanf()在循环外面获取的值。因此，在 while 循环的末尾还要使用一次 scanf()语句。这是处理类似问题的常用方法。

6. 下面是该程序的输出：

```
%s! C is cool!
! C is cool!
11
11
12
11
```

解释一下。第 1 个 printf()语句与下面的语句相同：

```
printf("%s! C is cool!\n","%s! C is cool!\n");
```

第 2 个 printf()语句首先把 num 递增为 11，然后打印该值。第 3 个 printf()语句打印 num 的值（值为 11）。第 4 个 printf()语句打印 n 当前的值（仍为 12），然后将其递减为 11。最后一个 printf()语句打印 num 的当前值（值为 11）。

7. 下面是该程序的输出：

```
SOS:4 4.00
```

表达式 c1 -c2 的值和'S' - '0'的值相同（其对应的 ASCII 值是 83 - 79）。

8. 把 1～10 打印在一行，每个数字占 5 列宽度，然后开始新的一行：

```
1 2 3 4 5 6 7 8 9 10
```

9. 下面是一个参考程序，假定字母连续编码，与 ASCII 中的情况一样。

```
#include <stdio.h>

int main(void)
{
    char ch = 'a';
    while (ch <= 'g')
         printf("%5c", ch++);
    printf("\n");
    return 0;
}
```

10. 下面是每个部分的输出：

a. `1  2`

注意，先递增 x 的值再比较。光标仍留在同一行。

b.

```
101
102
103
104
```

注意，这次 x 先比较后递增。在示例 a 和 b 中，x 都是在先递增后打印。另外还要注意，虽然第 2 个 printf()语句缩进了，但是这并不意味着它是 while 循环的一部分。因此，在 while 循环结束后，才会调用一次该 printf()语句。

c. `stuvw`

该例中，在第 1 次调用 printf()语句后才会递增 ch。

11. 这个程序有点问题。while 循环没有用花括号把两个缩进的语句括起来，只有 printf()是循环的一部分，所以该程序一直重复打印消息 COMPUTER BYTES DOG，直到强行关闭程序为止。

12. a. `x = x + 10;`

b. `x++; or ++x; or x = x + 1;`

c. `c = 2 * (a + b);`

d. c = a + 2* b;

13. a. x--; or --x; or x = x - 1;

b. m = n % k;

c. p = q / (b - a);

d. x = (a + b) / (c * d);

A.6　第 6 章复习题答案

1. 2，7，70，64，8，2。

2. 该循环的输出是：

   36 18 9 4 2 1

   如果 value 是 double 类型，即使 value 小于 1，循环的测试条件仍然为真。循环将一直执行，直到浮点数下溢生成 0 为止。另外，value 是 double 类型时，%3d 转换说明也不正确。

3. a. x > 5

   b. scanf("%lf",&x) != 1

   c. x == 5

4. a. scanf("%d", &x) == 1

   b. x != 5

   c. x >= 20

5. 第 4 行：应该是 list[10]。

   第 6 行：逗号改为分号。i 的范围应该是 0～9，不是 1～10。

   第 9 行：逗号改为分号。>=改成<=，否则，当 i 等于 1 时，该循环将成为无限循环。

   第 10 行：在第 10 行和第 11 行之间少了一个右花括号。该右花括号与第 7 行的左花括号配对，形成一个 for 循环块。然后在这个右花括号与最后一个右花括号之间，少了一行 return 0;。

   下面是一个正确的版本：

```
#include <stdio.h>
int main(void)
{                                       /* 第 3 行 */
    int i, j, list(10);                 /* 第 4 行 */

    for (i = 1, i <= 10, i++)           /* 第 6 行 */
    {                                   /* 第 7 行 */
        list[i] = 2*i + 3;              /* 第 8 行 */
        for (j = 1, j > = i, j++)       /* 第 9 行 */
            printf(" %d", list[j]);     /* 第 10 行 */
        printf("\n");                   /* 第 11 行 */
    }
    return 0;
}
```

6. 下面是一种方法：

```
#include <stdio.h>
int main(void)
```

```
{
    int col, row;

    for (row = 1; row <= 4; row++)
    {
        for (col = 1; col <= 8; col++)
            printf("$");
        printf("\n");
    }
    return 0;
}
```

7. a. `Hi! Hi! Hi! Bye! Bye! Bye! Bye! Bye!`

   b. `ACGM`（因为代码中把 `int` 类型值与 `char` 类型值相加，编译器可能警告会损失有效数字）

8. a. `Go west, youn`

   b. `Hp!xftu-!zpvo`

   c. `Go west, young`

   d. `$o west, youn`

9. 其输入如下：

```
31|32|33|30|31|32|33|
***
1
5
9
13

***
2 6
4 8
8 10

***
======
=====
====
===
==
```

10. a. `mint`

    b. `10` 个元素

    c. `double` 类型的值

    d. 第 `ii` 行正确，`mint[2]`是 `double` 类型的值，`&mingt[2]`是它在内存中的位置。

11. 因为第 `1` 个元素的索引是 `0`，所以循环的范围应该是 `0`～`SIZE - 1`，而不是 `1`～`SIZE`。但是，如果只是这样更改会导致赋给第 `1` 个元素的值是 `0`，不是 `2`。所以，应重写这个循环：

```
for (index = 0; index < SIZE; index++)
    by_twos[index] = 2 * (index + 1);
```

与此类似，第 `2` 个循环的范围也要更改。另外，应该在数组名后面使用数组索引：

```
for( index = 0; index < SIZE; index++)
    printf("%d ", by_twos[index]);
```

错误的循环条件会成为程序的定时炸弹。程序可能开始运行良好，但是由于数据被放在错误的位置，可能在某一时刻导致程序不能正常工作。

12. 该函数应声明为返回类型为 long，并包含一个返回 long 类型值的 return 语句。

13. 把 num 的类型强制转换成 long 类型，确保计算使用 long 类型而不是 int 类型。在 int 为 16 位的系统中，两个 int 类型值的乘积在返回之前会被截断为一个 int 类型的值，这可能会丢失数据。

```
long square(int num)
{
    return ((long) num) * num;
}
```

14. 输出如下：

```
1: Hi!
k = 1
k is 1 in the loop
Now k is 3
k = 3
k is 3 in the loop
Now k is 5
k = 5
k is 5 in the loop
Now k is 7
k = 7
```

A.7　第 7 章复习题答案

1. b 是 true。

2. a. number >= 90 && number < 100

   b. ch != 'q' && ch != 'k'

   c. (number >= 1 && number <= 9) && number != 5

   d. 可以写成!(number >= 1 && number <= 9)，但是 number < 1 || number > 9 更好理解。

3. 第 5 行：应该是 scanf("%d %d", &weight, &height);。不要忘记 scanf()中要用&。另外，这一行前面应该有提示用户输入的语句。

   第 9 行：测试条件中要表达的意思是(height < 72 && height > 64)。根据前面第 7 行中的测试条件，能到第 9 行的 height 一定小于 72，所以，只需要用表达式(height > 64)即可。但是，第 6 行中已经包含了 height > 64 这个条件，所以这里完全不必再判断，if else 应改成 else。

   第 11 行：条件冗余。第 2 个表达式（weight 不小于或不等于 300）和第 1 个表达式含义相同。只需用一个简单的表达式(weight > 300)即可。但是，问题不止于此。第 11 行是一个错误的 if，这行的 else if 与第 6 行的 if 匹配。但是，根据 if 的“最接近规则”，该 else if 应该与第 9 行的 else if 匹配。因此，在 weight 小于 100 且小于或等于 64 时到达第 11 行，而此时 weight 不可能超过 300。

   第 7 行～第 10 行：应该用花括号括起来。这样第 11 行就确定与第 6 行匹配。但是，如果把第 9 行的 else if 替换成简单的 else，就不需要使用花括号。

   第 13 行：应简化成 if (height > 48)。实际上，完全可以省略这一行。因为第 12 行已经测

试过该条件。

下面是修改后的版本：

```
#include <stdio.h>
int main(void)
{
    int weight, height; /* weight in lbs, height in inches */

    printf("Enter your weight in pounds and ");
    printf("your height in inches.\n");
    scanf("%d %d", &weight, &height);
    if (weight < 100 && height > 64)
        if (height >= 72)
            printf("You are very tall for your weight.\n");
        else
            printf("You are tall for your weight.\n");
    else if (weight > 300 && height < 48)
             printf(" You are quite short for your weight.\n");
    else
        printf("Your weight is ideal.\n");

    return 0;
}
```

4. a. 1。5 确实大于 2，表达式为真，即是 1。

   b. 0。3 比 2 大，表达式为假，即是 0。

   c. 1。如果第 1 个表达式为假，则第 2 个表达式为真，反之亦然。所以，只要一个表达式为真，整个表达式的结果即为真。

   d. 6。因为 6 > 2 为真，所以(6 > 2)的值为 1。

   e. 10。因为测试条件为真。

   f. 0。如果 x > y 为真，表达式的值就是 y > x，这种情况下它为假或 0。如果 x > y 为假，那么表达式的值就是 x > y，这种情况下为假。

5. 该程序打印以下内容：

   ```
   *#%*#%$#%*#%*#%$#%*#%*#%$#%*#%*#%
   ```

   无论怎样缩排，每次循环都会打印#，因为缩排并不能让 putchar('#');成为 if else 复合语句的一部分。

6. 程序打印以下内容：

   ```
   fat hat cat Oh no!
   hat cat Oh no!
   cat Oh no!
   ```

7. 第 5 行～第 7 行的注释要以*/结尾，或者把注释开头的/*换成//。表达式'a' <= ch >= 'z'应替换成 ch >= 'a' && ch <= 'z'。

   或者，包含 ctype.h 并使用 islower()，这种方法更简单，而且可移植性更高。顺带一提，虽然从 C 的语法方面看，'a' <= ch >= 'z'是有效的表达式，但是它的含义不明。因为关系运算符从左往右结合，该表达式被解释成('a' <= ch) >= 'z'。圆括号中的表达式的值不是 1 就是 0（真或假），然后判断该值是否大于或等于'z'的数值码。1 和 0 都不满足测试条件，所以整个表达式恒为 0（假）。在第 2 个测试表达式中，应该把||改成&&。另外，虽然!(ch< 'A')是有

效的表达式，而且含义也正确，但是用 ch >= 'A'更简单。这一行的'z'后面应该有两个圆括号。更简单的方法是使用 isuupper()。在 uc++;前面应该加一行 else。否则，每输入一个字符，uc 都会递增 1。另外，在 printf()语句中的格式化字符串应该用双引号括起来。下面是修改后的版本：

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char ch;
    int lc = 0; /*统计小写字母*/
    int uc = 0; /*统计大写字母*/
    int oc = 0; /*统计其他字母*/

    while ((ch = getchar()) != '#')
    {
        if (islower(ch))
            lc++;
        else if (isupper(ch))
            uc++;
        else
            oc++;
    }
    printf("%d lowercase, %d uppercase, %d other", lc, uc, oc);
    return 0;
}
```

8. 该程序将不停重复打印下面一行：

```
You are 65. Here is your gold watch.
```

问题出在这一行：if (age = 65)

这行代码把 age 设置为 65，使得每次迭代的测试条件都为真。

9. 下面是根据给定输入的运行结果：

```
q
Step 1
Step 2
Step 3
c
Step 1
h
Step 1
Step 3
b
Step 1
Done
```

注意，b 和#都可以结束循环。但是输入 b 会使得程序打印 step 1，而输入#则不会。

10. 下面是一种解决方案：

```
#include <stdio.h>
int main(void)
{
    char ch;
    while ((ch = getchar()) != '#')
```

```
    {
        if (ch != '\n')
        {
            printf("Step 1\n");
            if (ch == 'b')
                break;
            else if (ch != 'c')
            {
                if (ch != 'h')
                 printf("Step 2\n");
                printf("Step 3\n");
            }
        }
    }
    printf("Done\n");
    return 0;
}
```

A.8　第 8 章复习题答案

1. 表达式 putchar(getchar())使程序读取下一个输入字符并打印出来。getchar()的返回值是 putchar()的参数。但 getchar(putchar())是无效的表达式，因为 getchar()不需要参数，而 putchar()需要一个参数。

2. a．显示字符 H。

   b．如果系统使用 ASCII，则发出一声警报。

   c．把光标移至下一行的开始。

   d．退后一格。

3. count <essay >essayct 或者 count >essayct <essay

4. 都不是有效的命令。

5. EOF 是由 getchar()和 scanf()返回的信号（一个特殊值），表明函数检测到文件结尾。

6. a．输出是：If you qu

   注意，字符 I 与字符 i 不同。还要注意，没有打印 i，因为循环在检测到 i 之后就退出了。

   b．如果系统使用 ASCII，输出是：HJacrthjacrt

   while 的第 1 轮迭代中，为 ch 读取的值是 H。第 1 个 putchar()语句使用的 ch 的值是 H，打印完毕后，ch 的值加 1（现在是 ch 的值是 I）。然后到第 2 个 putchar()语句，因为是++ch，所以先递增 ch（现在 ch 的值是 J）再打印它的值。然后进入下一轮迭代，读取输入序列中的下一个字符（a），重复以上步骤。需要注意的是，两个递增运算符只在 ch 被赋值后影响它的值，不会让程序在输入序列中移动。

7. C 的标准 I/O 库把不同的文件映射为统一的流来统一处理。

8. 数值输入会跳过空格和换行符，但是字符输入不会。假设有下面的代码：

```
int score;
char grade;
printf("Enter the score.\n");
scanf("%s", %score);
```

```
printf("Enter the letter grade.\n");
grade = getchar();
```

如果输入分数 98，然后按下 Enter 键把分数发送给程序，其实还发送了一个换行符。这个换行符会留在输入序列中，成为下一个读取的值（grade）。如果在字符输入之前输入了数字，就应该在处理字符输入之前添加删除换行符的代码。

A.9　第 9 章复习题答案

1. 形式参数是定义在被调函数中的变量。实际参数是出现在函数调用中的值，该值被赋给形式参数。可以把实际参数视为在函数调用时初始化形式参数的值。

2. a. `void donut(int n)`

   b. `int gear(int t1, int t2)`

   c. `int guess(void)`

   d. `void stuff_it(double d, double *pd)`

3. a. `char n_to_char(int n)`

   b. `int digits(double x, int n)`

   c. `double * which(double * p1, double * p2)`

   d. `int random(void)`

4.

```
int sum(int a, int b)
{
    return a + b;
}
```

5. 用 double 替换 int 即可：

```
double sum(double a, double b)
{
    return a + b;
}
```

6. 该函数要使用指针：

```
void alter(int * pa, int * pb)
{
    int temp;
    temp = *pa + *pb;
    *pb = *pa - *pb;
    *pa = temp;
}
```

或者：

```
void alter(int * pa, int * pb)
{
    *pa += *pb;
    *pb = *pa - 2 * *pb;
}
```

7. 不正确。num 应声明在 salami()函数的参数列表中，而不是声明在函数体中。另外，把 count++改成 num++。

8. 下面是一种方案：

```
int largest(int a, int b, int c)
{
    int max = a;
    if (b > max)
        max = b;
    if (c > max)
        max = c;
    return max;
}
```

9. 下面是一个最小的程序，showmenu()和 getchoice()函数分别是 a 和 b 的答案。

```
#include <stdio.h>
/* 声明程序中要用到的函数 */
void showmenu(void);
int getchoice(int, int);
int main()
{
    int res;
    showmenu();
    while ((res = getchoice(1, 4)) != 4)
    {
        printf("I like choice %d.\n", res);
        showmenu();
    }
    printf("Bye!\n");
    return 0;
}
void showmenu(void)
{
    printf("Please choose one of the following:\n");
    printf("1) copy files          2) move files\n");
    printf("3) remove files        4) quit\n");
    printf("Enter the number of your choice:\n");
}
int getchoice(int low, int high)
{
    int ans;
    int good;
    good = scanf("%d", &ans);
    while (good == 1 && (ans < low || ans > high))
    {
        printf("%d is not a valid choice; try again\n", ans);
        showmenu();
        scanf("%d", &ans);
    }
    if (good != 1)
    {
        printf("Non-numeric input. ");
        ans = 4;
    }
    return ans;
}
```

A.10　第 10 章复习题答案

1．打印的内容如下：

```
8 8
4 4
0 0
2 2
```

2．数组 ref 有 4 个元素，因为初始化列表中的值是 4 个。

3．数组名 ref 指向该数组的首元素（整数 8）。表达式 ref + 1 指向该数组的第 2 个元素（整数 4）。++ref 不是有效的表达式，因为 ref 是一个常量，不是变量。

4．ptr 指向第 1 个元素，ptr + 2 指向第 3 个元素（即第 2 行的第 1 个元素）。

a．12 和 16。

b．12 和 14（初始化列表中，用花括号把 12 括起来，把 14 和 16 括起来，所以 12 初始化第 1 行的第 1 个元素，而 14 初始化第 2 行的第 1 个元素）。

5．ptr 指向第 1 行，ptr + 1 指向第 2 行。*ptr 指向第 1 行的第 1 个元素，而*(ptr + 1)指向第 2 行的第 1 个元素。

a．12 和 16。

b．12 和 14（同第 4 题，12 初始化第 1 行的第 1 个元素，而 14 初始化第 2 行的第 1 个元素）。

6．a．&grid[22][56]

b．&grid[22][0]或 grid[22]

（grid[22]是一个内含 100 个元素的一维数组，因此它就是首元素 grid[22][0]的地址。）

c．&grid[0][0]或 grid[0]或(int *) grid

（grid[0]是 int 类型元素 grid[0][0]的地址，grid 是内含 100 个元素的 grid[0]数组的地址。这两个地址的数值相同，但是类型不同，可以用强制类型转换把它们转换成相同的类型。）

7．a．int digits[10];

b．float rates[6];

c．int mat[3][5];

d．char * psa[20] ;

注意，[]比*的优先级高，所以在没有圆括号的情况下，psa 先与[20]结合，然后再与*结合。因此该声明与 char *(psa[20]);相同。

e．char (*pstr)[20];

注意

对第 e 小题而言，char *pstr[20];不正确。这会让 pstr 成为一个指针数组，而不是一个指向数组的指针。具体地说，如果使用该声明，pstr 就指向一个 char 类型的值（即数组的第 1 个成员），而 pstr + 1 则指向下一个字节。使用正确的声明，pstr 是一个变量，而不是一个数组名。而且 pstr + 1 指向起始字节后面的第 20 个字节。

8. a. `int sextet[6] = {1, 2, 4, 8, 16, 32};`

   b. `sextet[2]`

   c. `int lots[100] = { [99] = -1};`

   d. `int pots[100] = { [5] = 101, [10] = 101,101, 101, 101};`

9. 0～9

10. a. `rootbeer[2] = value;`有效。

    b. `scanf("%f", &rootbeer );`无效，rootbeer 不是 float 类型。

    c. `rootbeer = value;`无效，rootbeer 不是 float 类型。

    d. `printf("%f", rootbeer);`无效，rootbeer 不是 float 类型。

    e. `things[4][4] = rootbeer[3];`有效。

    f. `things[5] = rootbeer;`无效，不能用数组赋值。

    g. `pf = value;`无效，value 不是地址。

    h. `pf = rootbeer;`有效。

11. `int screen[800][600]` ；

12. a.

```
void process(double ar[], int n);
void processvla(int n, double ar[n]);
process(trots, 20);
processvla(20, trots);
```

    b.

```
void process2(short ar2[30], int n);
void process2vla(int n, int m, short ar2[n][m]);
process2(clops, 10);
process2vla(10, 30, clops);
```

    c.

```
void process3(long ar3[10][15], int n);
void process3vla(int n, int m,int k, long ar3[n][m][k]);
process3(shots, 5);
process3vla(5, 10, 15, shots);
```

13. a.

```
show( (int [4]) {8,3,9,2}, 4);
```

    b.

```
show2( (int [][3]){{8,3,9}, {5,4,1}}, 2);
```

A.11　第 11 章复习题答案

1. 如果希望得到一个字符串，初始化列表中应包含'\0'。当然，也可以用另一种语法自动添加空字符：

   `char name[] = "Fess";`

2.

```
See you at the snack bar.
ee you at the snack bar.
See you
e you
```

3.

```
y
my
mmy
ummy
Yummy
```

4. I read part of it all the way through.

5. a. Ho Ho Ho!!oH oH oH

   b. 指向 char 的指针（即，char *）。

   c. 第 1 个 H 的地址。

   d. *--pc 的意思是把指针递减 1，并使用储存在该位置上的值。--*pc 的意思是解引用 pc 指向的值，然后把该值减 1（例如，H 变成 G）。

   e. Ho Ho Ho!!oH oH o

注意

在两个! 之间有一个空字符，但是通常该字符不会产生任何打印的效果。

   f. while (*pc)检查 pc 是否指向一个空字符（即，是否指向字符串的末尾）。while 的测试条件中使用储存在指针指向位置上的值。

   while (pc - str)检查 pc 是否与 str 指向相同的位置（即，字符串的开头）。while 的测试条件中使用储存在指针指向位置上的值。

   g. 进入第 1 个 while 循环后，pc 指向空字符。进入第 2 个 while 循环后，它指向空字符前面的存储区（即，str 所指向位置前面的位置）。把该字节解释成一个字符，并打印这个字符。然后指针退回到前面的字节处。永远都不会满足结束条件(pc == str)，所以这个过程会一直持续下去。

   h. 必须在主调程序中声明 pr()：char * pr(char *);

6. 字符变量占用一个字节，所以 sign 占 1 字节。但是字符常量储存为 int 类型，意思是'$'通常占用 2 或 4 字节。但是实际上只使用 int 的 1 字节储存'$'的编码。字符串"$"使用 2 字节：一个字节储存'$'的编码，一个字节储存的'\0'编码。

7. 打印的内容如下：

```
How are ya, sweetie? How are ya, sweetie?
Beat the clock.
eat the clock.
Beat the clock. Win a toy.
Beat
chat
hat
at
t
t
at
How are ya, sweetie?
```

8. 打印的内容如下：

```
faavrhee
```

```
*le*on*sm
```

9. 下面是一种方案：

```
#include <stdio.h> // 提供 fgets()和 getchar()的原型
char * s_gets(char * st, int n)
{
    char * ret_val;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (*st != '\n' && *st != '\0')
            st++;
        if (*st == '\n')
            *st = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

10. 下面是一种方案：

```
int strlen(const char * s)
{
    int ct = 0;
    while (*s++)       // 或者 while (*s++ != '\0')
        ct++;
    return(ct);
}
```

11. 下面是一种方案：

```
#include <stdio.h>     // 提供 fgets()和 getchar()的原型
#include <string.h>    // 提供 strchr()的原型
char * s_gets(char * st, int n)
{
    char * ret_val;
    char * find;
    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        find = strchr(st, '\n');   // 查找换行符
        if (find)                  // 如果地址不是 NULL,
            *find = '\0';          // 在此处放置一个空字符
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

12. 下面是一种方案：

```
#include <stdio.h>     /* 提供 NULL 的定义 */
char * strblk(char * string)
{
```

```
    while (*string != ' ' && *string != '\0')
        string++;         /* 在第 1 个空白或空字符处停止 */
    if (*string == '\0')
        return NULL;      /* NULL 指空指针 */
    else
        return string;
}
```

下面是第 2 种方案，可以防止函数修改字符串，但是允许使用返回值改变字符串。表达式(char *)string 被称为“通过强制类型转换取消 const”。

```
#include <stdio.h>    /*提供 NULL 的定义*/
char * strblk(const char * string)
{
    while (*string != ' ' && *string != '\0')
        string++;         /*在第 1 个空白或空字符处停止*/
    if (*string == '\0')
        return NULL;      /* NULL 指空指针*/
    else
        return (char *)string;
}
```

13. 下面是一种方案：

```
/* compare.c -- 可行方案 */
#include <stdio.h>
#include <string.h> // 提供strcmp()的原型
#include <ctype.h>
#define ANSWER "GRANT"
#define SIZE 40
char * s_gets(char * st, int n);
void ToUpper(char * str);

int main(void)
{
    char try[SIZE];
    puts("Who is buried in Grant's tomb?");
    s_gets(try, SIZE);
    ToUpper(try);
    while (strcmp(try, ANSWER) != 0)
    {
        puts("No, that's wrong. Try again.");
        s_gets(try, SIZE);
        ToUpper(try);
    }
    puts("That's right!");
    return 0;
}

void ToUpper(char * str)
{
    while (*str != '\0')
    {
        *str = toupper(*str);
        str++;
    }
```

```
}

char * s_gets(char * st, int n)
{
    char * ret_val;
    int i = 0;

    ret_val = fgets(st, n, stdin);
    if (ret_val)
    {
        while (st[i] != '\n' && st[i] != '\0')
            i++;
        if (st[i] == '\n')
            st[i] = '\0';
        else
            while (getchar() != '\n')
                continue;
    }
    return ret_val;
}
```

A.12　第 12 章复习题答案

1. 自动存储类别；寄存器存储类别；静态、无链接存储类别。
2. 静态、无链接存储类别；静态、内部链接存储类别；静态、外部链接存储类别。
3. 静态、外部链接存储类别可以被多个文件使用。静态、内部链接存储类别只能在一个文件中使用。
4. 无链接。
5. 关键字 extern 用于声明中，表明该变量或函数已定义在别处。
6. 两者都分配了一个内含 100 个 int 类型值的数组。第 2 行代码使用 calloc()把数组中的每个元素都设置为 0。
7. 默认情况下，daisy 只对 main()可见，以 extern 声明的 daisy 才对 petal()、stem()和 root()可见。文件 2 中的 extern int daisy;声明使得 daisy 对文件 2 中的所有函数都可见。第 1 个 lily 是 main()的局部变量。petal()函数中引用的 lily 是错误的，因为两个文件中都没有外部链接的 lily。虽然文件 2 中有一个静态的 lily，但是它只对文件 2 可见。第 1 个外部 rose 对 root()函数可见，但是 stem()中的局部 rose 覆盖了外部的 rose。
8. 下面是程序的输出：

   ```
   color in main() is B
   color in first() is R
   color in main() is B
   color in second() is G
   color in main() is G
   ```

   first()函数没有使用 color 变量，但是 second()函数使用了。
9. a. 声明告诉我们，程序将使用一个变量 plink，该文件包含的函数都可以使用这个变量。calu_ct()函数的第 1 个参数是指向一个整数的指针，并假定它指向内含 n 个元素的数组。这里关键是要理解该程序不允许使用指针 arr 修改原始数组中的值。

   b. 不会。value 和 n 已经是原始数据的备份，所以该函数无法更改主调函数中相应的值。这些声

明的作用是防止函数修改 value 和 n 的值。例如，如果用 const 限定 n，就不能使用 n++表达式。

A.13　第 13 章复习题答案

1. 根据文件定义，应包含#include <stdio.h>。应该把 fp 声明为文件指针：FILE *fp;。要给 fopen()函数提供一种模式：fopen("gelatin","w")，或者"a"模式。fputs()函数的参数顺序应该反过来。输出字符串应该有一个换行符，提高可读性。fclose()函数需要一个文件指针，而不是一个文件名：fclose(fp);。下面是修改后的版本：

```
#include <stdio.h>
int main(void)
{
    FILE * fp;
    int k;
    fp = fopen("gelatin", "w");
    for (k = 0; k < 30; k++)
        fputs("Nanette eats gelatin.\n", fp);
    fclose(fp);
    return 0;
}
```

2. 如果可以打开的话，会打开与命令行第 1 个参数名相同名称的文件，并在屏幕上显示文件中的每个数字字符。

3. a. ch = getc(fp1);

   b. fprintf(fp2,"%c"\n",ch);

   c. putc(ch,fp2);

   d. fclose(fp1); /* 关闭 terky 文件 */

> **注意**
>
> fp1 用于输入操作，因为它识别以读模式打开的文件。与此类似，fp2 以写模式打开文件，所以常用于输出操作。

4. 下面是一种方案：

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv [])
{
    FILE * fp;
    double n;
    double sum = 0.0;
    int ct = 0;

    if (argc == 1)
        fp = stdin;
    else if (argc == 2)
    {
        if ((fp = fopen(argv[1], "r")) == NULL)
```

```
            {
                fprintf(stderr, "Can't open %s\n", argv[1]);
                exit(EXIT_FAILURE);
            }
        }
        else
        {
            fprintf(stderr, "Usage: %s [filename]\n", argv[0]);
            exit(EXIT_FAILURE);
        }
        while (fscanf(fp, "%lf", &n) == 1)
        {
            sum += n;
                ++ct;
        }
        if (ct > 0)
            printf("Average of %d values = %f\n", ct, sum / ct);
        else
            printf("No valid data.\n");

        return 0;
    }
```

5. 下面是一种方案：

```
#include <stdio.h>
#include <stdlib.h>
#define BUF 256
int has_ch(char ch, const char * line);
int main(int argc, char * argv [])
{
    FILE * fp;
    char ch;
    char line[BUF];

if (argc != 3)
    {
        printf("Usage: %s character filename\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    ch = argv[1][0];
    if ((fp = fopen(argv[2], "r")) == NULL)
    {
        printf("Can't open %s\n", argv[2]);
        exit(EXIT_FAILURE);
    }
    while (fgets(line, BUF, fp) != NULL)
    {
        if (has_ch(ch, line))
            fputs(line, stdout);
    }
    fclose(fp);
    return 0;
}
int has_ch(char ch, const char * line)
{
   while (*line)
```

```
        if (ch == *line++)
            return(1);
    return 0;
}
```

fgets()和 fputs()函数要一起使用，因为 fgets()会把按下 Enter 键的\n 留在字符串中，fputs()与 puts()不一样，不会添加一个换行符。

6. 二进制文件与文本文件的区别是，这两种文件格式对系统的依赖性不同。二进制流和文本流的区别包括是在读写流时程序执行的转换（二进制流不转换，而文本流可能要转换换行符和其他字符）。

7. a. 用 fprintf()储存 8238201 时，将其视为 7 个字符，保存在 7 字节中。用 fwrite()储存时，使用该数的二进制表示，将其储存为一个 4 字节的整数。

   b. 没有区别。两个函数都将其储存为一个单字节的二进制码。

8. 第 1 条语句是第 2 条语句的速记表示。第 3 条语句把消息写到标准错误上。通常，标准错误被定向到与标准输出相同的位置。但是标准错误不受标准输出重定向的影响。

9. 可以在以"r+"模式打开的文件中读写，所以该模式最合适。"a+"只允许在文件的末尾添加内容。"w+"模式提供一个空文件，丢弃文件原来的内容。

A.14　第 14 章复习题答案

1. 正确的关键是 struct，不是 structure。该结构模板要在左花括号前面有一个标记，或者在右花括号后面有一个结构变量名。另外，*togs 后面和模板结尾处都少一个分号。

2. 输出如下：

```
6 1
22 Spiffo Road
S p
```

3.

```
struct month {
    char name[10];
    char abbrev[4];
    int days;
    int monumb;
};
```

4.

```
struct month months[12] =
{
    { "January", "jan", 31, 1 },
    { "February", "feb", 28, 2 },
    { "March", "mar", 31, 3 },
    { "April", "apr", 30, 4 },
    { "May", "may", 31, 5 },
    { "June", "jun", 30, 6 },
    { "July", "jul", 31, 7 },
    { "August", "aug", 31, 8 },
    { "September", "sep", 30, 9 },
    { "October", "oct", 31, 10 },
    { "November", "nov", 30, 11 },
    { "December", "dec", 31, 12 }
};
```

5.

```
extern struct month months [];
int days(int month)
{
    int index, total;
    if (month < 1 || month > 12)
        return(-1); /* error signal */
    else
    {
        for (index = 0, total = 0; index < month; index++)
            total += months[index].days;
        return(total);
    }
}
```

注意，`index`比月数小1，因为数组下标从0开始。然后，用`index < month`代替`index <= month`。

6. a. 要包含`string.h`头文件，提供`strcpy()`的原型：

```
typedef struct lens { /* lens 描述 */
    float foclen;       /* 焦距长度，单位：mm */
    float fstop;        /* 孔径 */
    char brand[30];/* 品牌 */
} LENS;

LENS bigEye[10];
bigEye[2].foclen = 500;
bigEye[2].fstop = 2.0;
strcpy(bigEye[2].brand, "Remarkatar");
```

b. `LENS bigEye[10] = { [2] = {500, 2, "Remarkatar"} };`

7. a.

```
6
Arcturan
cturan
```

b. 使用结构名和指针：

```
deb.title.last
pb->title.last
```

c. 下面是一个版本：

```
#include <stdio.h>
#include "starfolk.h"       /* 让结构定义可用 */
void prbem (const struct bem * pbem )
{
    printf("%s %s is a %d-limbed %s.\n", pbem->title.first,
           pbem->title.last, pbem->limbs, pbem->type);
}
```

8. a. `willie.born`

b. `pt->born`

c. `scanf("%d", &willie.born);`

d. `scanf("%d", &pt->born);`

e. `scanf("%s", willie.name.lname);`

f. `scanf("%s", pt->name.lname);`

g. `willie.name.fname[2]`

h. `strlen(willie.name.fname) + strlen(willie.name.lname)`

9. 下面是一种方案：

```
struct car {
    char name[20];
     float hp;
     float epampg;
     float wbase;
     int year;
 };
```

10. 应该这样建立函数：

```
struct gas {
    float distance;
    float gals;
    float mpg;
};

struct gas mpgs(struct gas trip)
{
    if (trip.gals > 0)
        trip.mpg = trip.distance / trip.gals;
    else
        trip.mpg = -1.0;
    return trip;
}

void set_mpgs(struct gas * ptrip)
{
    if (ptrip->gals > 0)
        ptrip->mpg = ptrip->distance / ptrip->gals;
    else
        ptrip->mpg = -1.0;
}
```

注意，第 1 个函数不能直接改变其主调程序中的值，所以必须用返回值才能传递信息。

```
struct gas idaho = {430.0, 14.8};   // 设置前两个成员
idaho = mpgs(idaho);                // 重置数据结构
```

但是，第 2 个函数可以直接访问最初的结构：

```
struct gas ohio = {583, 17.6};      //设置前两个成员
set_mpgs(&ohio);                    // 设置第 3 个成员
```

11. `enum choices {no, yes, maybe};`

12. `char * (*pfun)(char *, char);`

13.

```
double sum(double, double);
double diff(double, double);
double times(double, double);
double divide(double, double);
double (*pf1[4])(double, double) = {sum, diff, times, divide};
```

或者用更简单的形式，把代码中最后一行替换成：

```
typedef double (*ptype) (double, double);
ptype pfl[4] = {sum,diff, times, divide};
```

调用 diff()函数：

```
pf1[1](10.0, 2.5);     // 第 1 种表示法
(*pf1[1])(10.0, 2.5);  // 等价表示法
```

A.15　第 15 章复习题答案

1. a. 00000011

   b. 00001101

   c. 00111011

   d. 01110111

2. a. 21, 025, 0x15

   b. 85, 0125, 0x55

   c. 76, 0114, 0x4C

   d. 157, 0235, 0x9D

3. a. 252

   b. 2

   c. 7

   d. 7

   e. 5

   f. 3

   g. 28

4. a. 255

   b. 1 (not false is true)

   c. 0

   d. 1 (true and true is true)

   e. 6

   f. 1 (true or true is true)

   g. 40

5. 掩码的二进制是 1111111；十进制是 127；八进制是 0177；十六进制是 0x7F。

6. bitval * 2 和 bitval << 1 都把 bitval 的当前值增加一倍，它们是等效的。但是 mask += bitval 和 mask |= bitval 只有在 bitval 和 mask 没有同时打开的位时效果才相同。例如，2 | 4 得 6，但是 3 | 6 也得 6。

7. a.

```
struct tb_drives {
    unsigned int diskdrives   : 2;
    unsigned int              : 1;
    unsigned int cdromdrives  : 2;
    unsigned int              : 1;
    unsigned int harddrives   : 2;
};
```

b.

```
struct kb_drives {
    unsigned int harddrives   : 2;
    unsigned int              : 1;
    unsigned int cdromdrives  : 2;
    unsigned int              : 1;
    unsigned int diskdrives   : 2;
};
```

A.16　第 16 章复习题答案

1. a. dist = 5280 * miles;有效。

   b. plort = 4 * 4 + 4;有效。但是如果用户需要的是 4 * (4 + 4)，则应该使用#define POD (FEET + FEET)。

   c. nex = = 6;;无效（如果两个等号之间没有空格，则有效，但是没有意义）。显然，用户忘记了在编写预处理器代码时不用加=。

   d. y = y + 5;有效。berg = berg + 5 * lob;有效，但是可能得不到想要的结果。est = berg + 5/y + 5;有效，但是可能得不到想要的结果。

2. #define NEW(X) ((X) + 5)

3. #define MIN(X,Y) ((X) < (Y) ? (X) : (Y))

4. #define EVEN_GT(X,Y) ((X) > (Y) && (X) % 2 == 0 ? 1 : 0)

5. #define PR(X,Y) printf(#X " is %d and " #Y " is %d\n", X,Y)

   （因为该宏中没有运算符（如，乘法）作用于 X 和 Y，所以不需要使用圆括号。）

6. a. #define QUARTERCENTURY 25

   b. #define SPACE ' '

   c. #define PS() putchar(' ')或#define PS() putchar(SPACE)

   d. #define BIG(X) ((X) + 3)

   e. #define SUMSQ(X,Y) ((X)*(X) + (Y)*(Y))

7. 试试这样：#define P(X) printf("name: "#X"; value: %d; address: %p\n", X, &X)

   （如果你的实现无法识别地址专用的%p 转换说明，可以用%u 或%lu 代替。）

8. 使用条件编译指令。一种方法是使用#ifndef：

```
#define _SKIP_ /* 如果不需要跳过代码，则删除这条指令 */
#ifndef _SKIP_
/* 需要跳过的代码 */
#endif
```

9.

```
#ifdef PR_DATE
    printf("Date = %s\n", _ _DATE_ _);
#endif
```

10. 第 1 个版本返回 x*x，这只是返回了 square()的 double 类型值。例如，square(1.3)会返回 1.69。第 2 个版本返回 (int)(x*x)，计算结果被截断后返回。但是，由于该函数的返回类型是 double，int 类型的值将被升级为 double 类型的值，所以 1.69 将先被转换成 1，然后

被转换成 1.00。第 3 个版本返回(int)(x*x+0.5)。加上 0.5 可以让函数把结果四舍五入至与原值最接近的值，而不是简单地截断。所以，1.69+0.5 得 2.19，然后被截断为 2，然后被转换成 2.00；而 1.44+0.5 得 1.94，被截断为 1，然后被转换成 1.00。

11. 这是一种方案：#define BOOL(X) _Generic((X), _Bool : "boolean", default : "not boolean")

12. 应该把 argv 参数声明为 char *argv[]类型。命令行参数被储存为字符串，所以该程序应该先把 argv[1]中的字符串转换成 double 类型的值。例如，用 stdlib.h 库中的 atof()函数。程序中使用了 sqrt()函数，所以应包含 math.h 头文件。程序在求平方根之前应排除参数为负的情况（检查参数是否大于或等于 0）。

13. a. qsort((void *)scores, (size_t) 1000, sizeof (double), comp);

    b. 下面是一个比较使用的比较函数：

```
int comp(const void * p1, const void * p2)
{
    /* 要用指向 int 的指针来访问值 */
    /* 在 C 中是否进行强制类型转换都可以，在 C++中必须进行强制类型转换 */
    const int * a1 = (const int *) p1; const int * a2 = (const int *)
 p2;
    if (*a1 > *a2)
        return -1;
    else if (*a1 == *a2)
        return 0;
    else
        return 1;
}
```

14. a. 函数调用应该类似：memcpy(data1, data2, 100 * sizeof(double));

    b. 函数调用应该类似：memcpy(data1, data2 + 200 , 100 * sizeof(double));

A.17　第 17 章复习题答案

1. 定义一种数据类型包括确定如何储存数据，以及设计管理该数据的一系列函数。
2. 因为每个结构包含下一个结构的地址，但是不包含上一个结构的地址，所以这个链表只能沿着一个方向遍历。可以修改结构，在结构中包含两个指针，一个指向上一个结构，一个指向下一个结构。当然，程序也要添加代码，在每次新增结构时为这些指针赋正确的地址。
3. ADT 是**抽象数据类型**，是对一种类型属性集和可以对该类型进行的操作的正式定义。ADT 应该用一般语言表示，而不是用某种特殊的计算机语言，而且不应该包含实现细节。
4. **直接传递变量的优点**：该函数查看一个队列，但是不改变其中的内容。直接传递队列变量，意味着该函数使用的是原始队列的副本，这保证了该函数不会更改原始的数据。直接传递变量时，不需要使用地址运算符或指针。

   直接传递变量的缺点：程序必须分配足够的空间储存整个变量，然后拷贝原始数据的信息。如果变量是一个大型结构，用这种方法将花费大量的时间和内存空间。

   传递变量地址的优点：如果待传递的变量是大型结构，那么传递变量的地址和访问原始数据会更快，所需的内存空间更少。

   传递变量地址的缺点：必须记得使用地址运算符或指针。在 K&R C 中，函数可能会不小心改变原

始数据，但是用 ANSI C 中的 const 限定符可以解决这个问题。

5．a.

类型名：　　　　栈

类型属性：　　　可以储存有序项

类型操作：　　　初始化栈为空

确定栈是否为空

确定栈是否已满

从栈顶添加项（压入项）

从栈顶删除项（弹出项）

b．下面以数组形式实现栈，但是这些信息只影响结构定义和函数定义的细节，不会影响函数原型的接口。

```
/* stack.h -- 栈的接口 */
#include <stdbool.h>
/* 在这里插入 Item 类型 */
/* 例如： typedef int Item; */

#define MAXSTACK 100

typedef struct stack
{
    Item items[MAXSTACK];   /* 储存信息         */
    int top;                /* 第 1 个空位的索引 */
} Stack;

/* 操作：         初始化栈                                    */
/* 前提条件：     ps 指向一个栈                               */
/* 后置条件：     该栈被初始化为空                            */
void InitializeStack(Stack * ps);

/* 操作：         检查栈是否已满                              */
/* 前提条件：     ps 指向之前已被初始化的栈                   */
/* 后置条件：     如果栈已满，该函数返回 true；否则，返回 false      */
bool FullStack(const Stack * ps);

/* 操作：         检查栈是否为空                              */
/* 前提条件：     ps 指向之前已被初始化的栈                   */
/* 后置条件：     如果栈为空，该函数返回 true；否则，返回 false      */
bool EmptyStack(const Stack *ps);

/* 操作：         把项压入栈顶                                */
/* 前提条件：     ps 指向之前已被初始化的栈                   */
/*                item 是待压入栈顶的项                       */
/* 后置条件：     如果栈不满，把 item 放在栈顶，该函数返回 ture；    */
/*                否则，栈不变，该函数返回 false              */
bool Push(Item item, Stack * ps);

/* 操作：         从栈顶删除项                                */
/* 前提条件：     ps 指向之前已被初始化的栈                   */
/* 后置条件：     如果栈不为空，把栈顶的 item 拷贝到*pitem，      */
```

```
/*          删除栈项的 item，该函数返回 ture；                    */
/*          如果该操作后栈中没有项，则重置该栈为空。              */
/*          如果删除操作之前栈为空，栈不变，该函数返回 false      */
bool Pop(Item *pitem, Stack * ps);
```

6．比较所需的最大次数如下：

项	顺序查找	二分查找
3	3	2
1023	1023	10
65535	65535	16

7．见图 A.1。

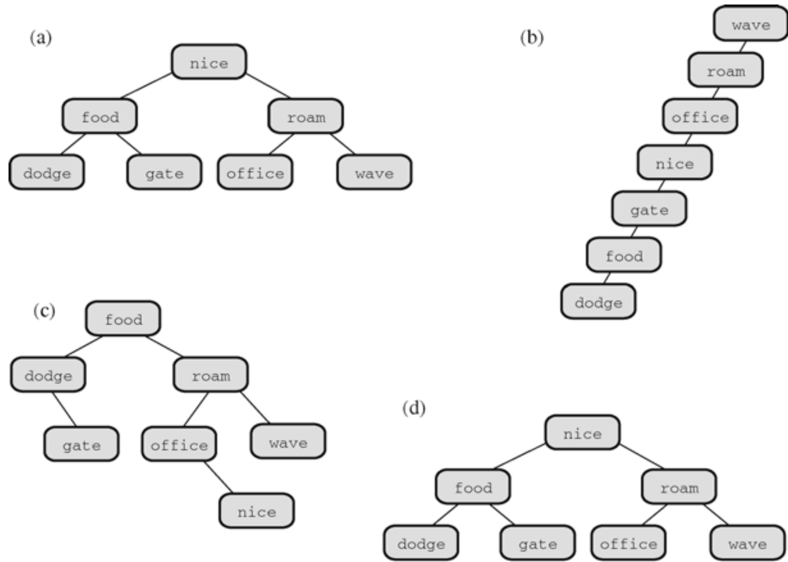


图 A.1　单词的二分查找树

8．见图 A.2。

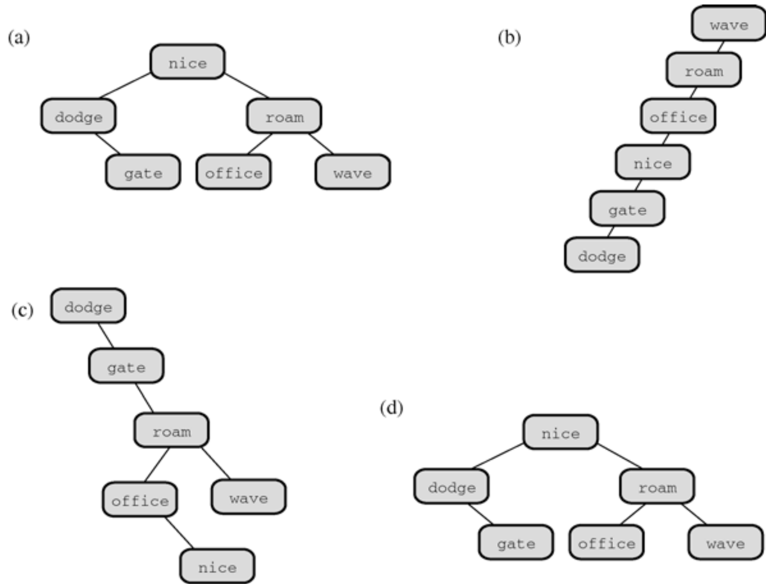


图 A.2　删除项后的单词二分查找树

附录 B 参考资料

本书这部分总结了 C 语言的基本特性和一些特定主题的详细内容，包括以下 9 个部分。

- 参考资料 I：补充阅读
- 参考资料 II：C 运算符
- 参考资料 III：基本类型和存储类别
- 参考资料 IV：表达式、语句和程序流
- 参考资料 V：新增了 C99 和 C11 的标准 ANSI C 库
- 参考资料 VI：扩展的整数类型
- 参考资料 VII：扩展的字符支持
- 参考资料 VIII：C99/C11 数值计算增强
- 参考资料 IX：C 与 C++的区别

B.1 参考资料 I：补充阅读

如果想了解更多 C 语言和编程方面的知识，下面提供的资料会对你有所帮助。

B.1.1 在线资源

C 程序员帮助建立了互联网，而互联网可以帮助你学习 C。互联网时刻都在发展、变化，这里所列的资源只是在撰写本书时可用的资源。当然，你可以在互联网中找到其他资源。

如果有一些与 C 语言相关的问题或只是想扩展你的知识，可以浏览 C FAQ（常见问题解答）的站点：

c-faq.com

但是，这个站点的内容主要涵盖到 C89。

如果对 C 库有疑问，可以访问这个站点获得信息：www.acm.uiuc .edu/webmonkeys/book/c_guide/index.html。

这个站点全面讨论指针：pweb.netcom.com/～tjensen/ptr/pointers.htm。

还可以使用谷歌和雅虎的搜索引擎，查找相关文章和站点：

www.google.com
search.yahoo.com
www.bing.com

可以使用这些站点中的高级搜索特性来优化你要搜索的内容。例如，尝试搜索 C 教程。

你可以通过新闻组（*newsgroup*）在网上提问。通常，新闻组阅读程序通过你的互联网服务提供商提供的账号访问新闻组。另一种访问方法是在网页浏览器中输入这个地址：http://groups.google.com。

你应该先花时间阅读新闻组，了解它涵盖了哪些主题。例如，如果你对如何使用 C 语言完成某事有疑问，可以试试这些新闻组：

comp.lang.c
comp.lang.c.moderated

可以在这里找到愿意提供帮助的人。你所提的问题应该与标准 C 语言相关，不要在这里询问如何在 UNIX 系统中获得无缓冲输入之类的问题。特定平台都有专门的新闻组。最重要的是，不要询问他们如何解决家庭作业中的问题。

如果对 C 标准有疑问，试试这个新闻组：comp.std.c。但是，不要在这里询问如何声明一个指向三维数组的指针，这类问题应该到另一个新闻组：comp.lang.c。

最后，如果对 C 语言的历史感兴趣，可以浏览下 C 创始人 Dennis Ritchie 的站点，其中 1993 年中有一篇文章介绍了 C 的起源和发展：cm.bell-labs.com/cm/cs/who/dmr/chist.html。

B.1.2　C 语言书籍

Feuer, Alan R. ***The C Puzzle Book***, *Revised Printing*. Upper Saddle River, NJ: Addison-WesleyProfessional, 1998。这本书包含了许多程序，可以用来学习，推测这些程序应输出的内容。预测输出对测试和扩展 C 的理解很有帮助。本书也附有答案和解释。

Kernighan, Brian W. and Dennis M. Ritchie. ***The C Programming Language***, Second Edition .Englewood Cliffs, NJ: Prentice Hall, 1988。第 1 本 C 语言书的第 2 版（注意，作者 Dennis Ritchie 是 C 的创始者）。本书的第 1 版给出了 K&R C 的定义，许多年来它都是非官方的标准。第 2 版基于当时的 ANSI 草案进行了修订，在编写本书时该草案已成为了标准。本书包含了许多有趣的例子，但是它假定读者已经熟悉了系统编程。

Koenig, Andrew. ***C Traps and Pitfalls*** . Reading, MA: Addison-Wesley, 1989。本书的中文版《C 陷阱与缺陷》已由人民邮电出版社出版。

Summit, Steve. ***C Programming FAQs*** . Reading, MA: Addison-Wesley, 1995。这本书是互联网 FAQ 的延伸阅读版本。

B.1.3　编程书籍

Kernighan, Brian W. and P.J. Plauger. ***The Elements of Programming Style***, Second Edition . NewYork: McGraw-Hill, 1978。这本短小精悍的绝版书籍，历经岁月却无法掩盖其真知灼见。书中介绍了要编写高效的程序，什么该做，什么不该做。

Knuth, Donald E. ***The Art of Computer Programming***, **第 1 卷（基本算法）**，Third Edition . Reading, MA: Addison-Wesley, 1997。这本经典的标准参考书非常详尽地介绍了数据表示和算法分析。第 2 卷（半数学算法，1997）探讨了伪随机数。第 3 卷（排序和搜索，1998）介绍了排序和搜索，以伪代码和汇编语言的形式给出示例。

Sedgewick, Robert. ***Algorithms in C***, Parts 1-4: ***Fundamentals, Data Structures, Sorting, Searching***,Third Edition. Reading, MA: Addison-Wesley Professional, 1997。顾名思义，这本书介绍了数据结构、排序和搜索。本书中文版《C 算法（第 1 卷）基础、数据结构、排序和搜索（第 3 版）》已由人民邮电出版社出版。

B.1.4　参考书籍

Harbison, Samuel P. and Steele, Guy L. C: ***A Reference Manual***, Fifth Edition. Englewood Cliffs,NJ: Prentice Hall, 2002。这本参考手册介绍了 C 语言的规则和大多数标准库函数。它结合了 C99，提供了许多例子.《C 语言参考手册（第 5 版）（英文版）》已由人民邮电出版社出版。

Plauger, P.J. ***The Standard C Library*** . Englewood Cliffs, NJ: Prentice Hall, 1992。这本大型的参考手册介

绍了标准库函数，比一般的编译器手册更详尽。

The International C Standard . ISO/IEC 9899:2011。在撰写本书时，可以花 285 美元从 www.ansi.org 下载该标准的电子版，或者花 238 欧元从 IEC 下载。别指望通过这本书学习 C 语言，因为它并不是一本学习教程。这是一句有代表性的话，可见一斑：“如果在一个翻译单元中声明一个特定标识符多次，在该翻译单元中都可见，那么语法可根据上下文无歧义地引用不同的实体”。

B.1.5　C++书籍

Prata, Stephen. ***C++ Primer Plus***, Sixth Edition . Upper Saddle River, NJ: Addison-Wesley, 2012。本书介绍了 C++语言（C++11 标准）和面向对象编程的原则。

Stroustrup, Bjarne. ***The C++ Programming Language***, Fourth Edition. Reading, MA: Addison-Wesley, 2013。本书由 C++的创始人撰写，介绍了 C++11 标准。

B.2　参考资料 II：C 运算符

C 语言有大量的运算符。表 B.2.1 按优先级从高至低的顺序列出了 C 运算符，并给出了其结合性。除非特别指明，否则所有运算符都是二元运算符（需要两个运算对象）。注意，一些二元运算符和一元运算符的表示符号相同，但是其优先级不同。例如，*（乘法运算符）和*（间接运算符）。表后面总结了每个运算符的用法。

表 B.2.1　C 运算符

运算符（优先级从高至低）	结合律
++（后缀）　--（后缀）　()（函数调用） []　{}　（复合字面量）　.　->	从左往右
++（前缀）　--（前缀）　-　+　～　! *（解引用）&（取址） sizeof　_Alignof(类型名)（本栏都是一元运算符）	从右往左
(类型名)	从右往左
*　/　%	从左往右
+　-（都是二元运算符）	从左往右
<<>>	从左往右
<><=　>=	从左往右
==　!=	从左往右
&	从左往右
^	从左往右
|	从左往右
&&	从左往右
||	从左往右
?:（条件表达式）	从右往左
=　*=　/=　+=　-=　<<=　>>=　&=　|=　^=	从右往左
,（逗号运算符）	从左往右

B.2.1　算术运算符

\+ 把右边的值加到左边的值上。

\+ 作为一元运算符，生成一个大小和符号都与右边值相同的值。

\- 从左边的值中减去右边的值。

\- 作为一元运算符，生成一个与右边值大小相等符号相反的值。

* 把左边的值乘以右边的值。

/ 把左边的值除以右边的值；如果两个运算对象都是整数，其结果要被截断。

% 得左边值除以右边值时的余数

++ 把右边变量的值加 1（前缀模式），或把左边变量的值加 1（后缀模式）。

-- 把右边变量的值减 1（前缀模式），或把左边变量的值减 1（后缀模式）。

B.2.2　关系运算符

下面的每个运算符都把左边的值与右边的值相比较。

<　　小于

<=　　小于或等于

==　　等于

>=　　大于或等于

>　　大于

!=　　不等于

关系表达式

简单的关系表达式由关系运算符及其两侧的运算对象组成。如果关系为真，则关系表达式的值为 1；如果关系为假，则关系表达式的值为 0。下面是两个例子：

5 > 2 关系为真，整个表达式的值为 1。

(2 + a) == a 关系为假，整个表达式的值为 0。

B.2.3　赋值运算符

C 语言有一个基本赋值运算符和多个复合赋值运算符。=运算符是基本的形式：

= 把它右边的值赋给其左边的左值。

下面的每个赋值运算符都根据它右边的值更新其左边的左值。我们使用 R-H 表示右边，L-R 表示左边。

+= 把左边的变量加上右边的量，并把结果储存在左边的变量中。

-= 从左边的变量中减去右边的量，并把结果储存在左边的变量中。

*= 把左边的变量乘以右边的量，并把结果储存在左边的变量中。

/= 把左边的变量除以右边的量，并把结果储存在左边的变量中。

%= 得到左边量除以右边量的余数，并把结果储存在左边的变量中。

&= 把 L-H & R-H 的值赋给左边的量，并把结果储存在左边的变量中。

|= 把 L-H | R-H 的值赋给左边的量，并把结果储存在左边的变量中。

^= 把 L-H ^ R-H 的值赋给左边的量，并把结果储存在左边的变量中。

>>= 把 L-H >> R-H 的值赋给左边的量，并把结果储存在左边的变量中。

<<= 把 L-H << R-H 的值赋给左边的量，并把结果储存在左边的变量中。

示例

rabbits *= 1.6;与 rabbits = rabbits * 1.6 效果相同。

B.2.4 逻辑运算符

逻辑运算符通常以关系表达式作为运算对象。!运算符只需要一个运算对象，其他运算符需要两个运算对象，运算符左边一个，右边一个。

&& 逻辑与

|| 逻辑或

! 逻辑非

1. 逻辑表达式

当且仅当两个表达式都为真时，expresson1 && expresson 2 的值才为真。

两个表达式中至少有一个为真时，expresson 1 && expresson 2 的值就为真。

如果 expresson 的值为假，则!expresson 为真，反之亦然。

2. 逻辑表达式的求值顺序

逻辑表达式的求值顺序是从左往右。当发现可以使整个表达式为假的条件时立即停止求值。

3. 示例

6 > 2 && 3 == 3 为真。

!(6 > 2 && 3 == 3) 为假。

x != 0 && 20/x < 5 只有在 x 是非零时才会对第 2 个表达式求值。

B.2.5 条件运算符

?:有 3 个运算对象，每个运算对象都是一个表达式：*expression1 ? expression2 : expression3*

如果 expression1 为真，则整个表达式的值等于 expression2 的值；否则，等于 expression3 的值。

示例

(5 > 3) ? 1 : 2 的值为 1。

(3 > 5) ? 1 : 2 的值为 2。

(a > b) ? a : b 的值是 a 和 b 中较大者

B.2.6 与指针有关的运算符

&是地址运算符。当它后面是一个变量名时，&给出该变量的地址。

*是间接或解引用运算符。当它后面是一个指针时，*给出储存在指针指向地址中的值。

示例

&nurse 是变量 nurse 的地址：

```
nurse = 22;
ptr = &nurse; /* 指向 nurse 的指针 */
val = *ptr;
```

以上代码的效果是把 22 赋给 val。

B.2.7　符号运算符

-是负号，反转运算对象的符号。

+ 是正号，不改变运算对象的符号。

B.2.8　结构和联合运算符

结构和联合使用一些运算符标识成员。成员运算符与结构和联合一起使用，间接成员运算符与指向结构或联合的指针一起使用。

1. 成员运算符

成员运算符（.）与结构名或联合名一起使用，指定结构或联合中的一个成员。如果 name 是一个结构名，member 是该结构模板指定的成员名，那么 name.member 标识该结构中的这个成员。name.member 的类型就是被指定 member 的类型。在联合中也可以用相同的方式使用成员运算符。

示例

```
struct {
    int code;
    float cost;
} item;
item.code = 1265;
```

上面这条语句把 1265 赋给结构变量 item 的成员 code。

2. 间接成员运算符（或结构指针运算符）

间接成员运算符（->）与一个指向结构或联合的指针一起使用，标识该结构或联合的一个成员。假设 ptrstr 是一个指向结构的指针，member 是该结构模板指定的成员，那么 ptrstr->member 标识了指针所指向结构的这个成员。在联合中也可以用相同的方式使用间接成员运算符。

示例

```
struct {
    int code;
    float cost;
} item, * ptrst;
ptrst = &item;
ptrst->code = 3451;
```

以上程序段把 3451 赋给结构 item 的成员 code。下面 3 种写法是等效的：

```
ptrst->code   item.code    (*ptrst).code
```

B.2.9　按位运算符

下面所列除了～，都是按位运算符。

～ 是一元运算符，它通过翻转运算对象的每一位得到一个值。

& 是逻辑与运算符，只有当两个运算对象中对应的位都为 1 时，它生成的值中对应的位才为 1。

| 是逻辑或运算符，只要两个运算对象中对应的位有一位为 1，它生成的值中对应的位就为 1。

^ 是按位异或运算符，只有两个运算对象中对应的位中只有一位为 1（不能全为 1），它生成的值中对应的位才为 1。

<< 是左移运算符，把左边运算对象中的位向左移动得到一个值。移动的位数由该运算符右边的运算对象确定，空出的位用 0 填充。

>> 是右移运算符，把左边运算对象中的位向右移动得到一个值。移动的位数由该运算符右边的运算对象确定，空出的位用 0 填充。

示例

假设有下面的代码：

```
int x = 2;
int y = 3;
```

x & y 的值为 2，因为 x 和 y 的位组合中，只有第 1 位均为 1。而 y << x 的值为 12，因为在 y 的位组合中，3 的位组合向左移动两位，得到 12。

B.2.10　混合运算符

sizeof 给出它右边运算对象的大小，单位是 char 的大小。通常，char 类型的大小是 1 字节。运算对象可以圆括号中的类型说明符，如 sizeof(float)，也可以是特定的变量名、数组名等，如 sizeof foo。sizeof 表达式的类型是 size_t。

_Alignof（C11）给出它的运算对象指定类型的对齐要求。一些系统要求以特定值的倍数在地址上储存特定类型，如 4 的倍数。这个整数就是对齐要求。

（类型名）是强制类型转换运算符，它把后面的值转换成圆括号中关键字指定的类型。例如，(float)9 把整数 9 转换成浮点数 9.0。

, 是逗号运算符，它把两个表达式链接成一个表达式，并保证先对最左端的表达式求值。整个表达式的值是最右边表达式的值。该运算符通常在 for 循环头中用于包含更多的信息。

示例

```
for (step = 2, fargo = 0; fargo < 1000; step *= 2)
     fargo += step;
```

B.3　参考资料 III：基本类型和存储类别

B.3.1　总结：基本数据类型

C 语言的基本数据类型分为两大类：整数类型和浮点数类型。不同的种类提供了不同的范围和精度。

1. 关键字

创建基本数据类型要用到 8 个关键字：int、long、short、unsigned、char、float、double、signed（ANSI C）。

2. 有符号整数

有符号整数可以具有正值或负值。

int 是所有系统中基本整数类型。

long 或 long int 可储存的整数应大于或等于 int 可储存的最大数；long 至少是 32 位。

short 或 short int 整数应小于或等于 int 可储存的最大数；short 至少是 16 位。通常，long 比 short 大。例如，在 PC 中的 C DOS 编译器提供 16 位的 short 和 int、32 位的 long。这完全取决于系统。

C99 标准提供了 long long 类型，至少和 long 一样大，至少是 64 位。

3. 无符号整数

无符号整数只有 0 和正值，这使得该类型能表示的正数范围更大。在所需的类型前面加上关键字 unsigned：unsigned int、unsigned long、unsigned short、unsigned long long。单独的 unsigned 相当于 unsigned int。

4. 字符

字符是如 A、&、+这样的印刷符号。根据定义，char 类型的变量占用 1 字节的内存。过去，char 类型的大小通常是 8 位。然而，C 在处理更大的字符集时，char 类型可以是 16 位，或者甚至是 32 位。

这种类型的关键字是 char。一些实现使用有符号的 char，但是其他实现使用无符号的 char。ANSI C 允许使用关键字 signed 和 unsigned 指定所需类型。从技术层面上看，char、unsigned char 和 signed char 是 3 种不同的类型，但是 char 类型与其他两种类型的表示方法相同。

5. 布尔类型（C99）

_Bool 是 C99 新增的布尔类型。它一个无符号整数类型，只能储存 0（表示假）或 1（表示真）。包含 stdbool.c 头文件后，可以用 bool 表示_Bool、ture 表示 1、false 表示 0，让代码与 C++兼容。

6. 实浮点数和复浮点数类型

C99 识别两种浮点数类型：实浮点数和复浮点数。浮点类型由这两种类型构成。

实浮点数可以是正值或负值。C 识别 3 种实浮点类型。

float 是系统中的基本浮点类型。它至少可以精确表示 6 位有效数字，通常 float 为 32 位。

double（可能）表示更大的浮点数。它能表示比 float 更多的有效数字和更大的指数。它至少能精确表示 10 位有效数字。通常，double 为 64 位。

long double（可能）表示更大的浮点数。它能表示比 double 更多的有效数字和更大的指数。

复数由两部分组成：实部和虚部。C99 规定一个复数在内部用一个有两个元素的数组表示，第 1 个元素表示实部，第 2 个元素表示虚部。有 3 种复浮点数类型。

float _Complex 表示实部和虚部都是 float 类型的值。

double _Complex 表示实部虚部都是 double 类型的值。

long double _Complex 表示实部和虚部都是 long double 类型的值。

每种情况，前缀部分的类型都称为**相应的实数类型**（*corresponding real type*）。例如，double 是 double _Complex 相应的实数类型。

C99 中，复数类型在独立环境中是可选的，这样的环境中不需要操作系统也可运行 C 程序。在 C11 中，复数类型在独立环境和主机环境都是可选的。

有 3 种虚数类型。它们在独立环境中和主机环境中（C 程序在一种操作系统下运行的环境）都是可选的。虚数只有虚部。这 3 种类型如下。

float _Imaginary 表示虚部是 float 类型的值。

double _Imaginary 表示虚部是 double 类型的值。

long double _Imaginary 表示虚部是 long double 类型的值。

可以用实数和 I 值来初始化复数。I 定义在 complex.h 头文件中，表示 i（即-1 的平方根）。

```
#include <complex.h>              // I 定义在该头文件中
double _Complex z = 3.0;          // 实部 = 3.0，虚部 = 0
double _Complex w = 4.0 * I;      // 实部 = 0.0，虚部 = 4.0
double Complex u = 6.0 – 8.0 * I; //实部= 6.0，虚部 = -8.0
```

前面章节讨论过，complex.h 库包含一些返回复数实部和虚部的函数。

B.3.2 总结：如何声明一个简单变量

1. 选择所需的类型。
2. 选择一个合适的变量名。
3. 使用这种声明格式：*type-specifiervariable-name;*

   type-specifier 由一个或多个类型关键字组成，下面是一些例子：

   ```
   int erest;
   unsigned short cash;
   ```
4. 声明多个同类型变量时，使用逗号分隔符隔开各变量名：

   ```
   char ch, init, ans;
   ```
5. 可以在声明的同时初始化变量：

   ```
   float mass = 6.0E24;
   ```

总结：存储类别

关键字：auto、extern、static、register、_Thread_local（C11）

一般注解：

变量的存储类别取决于它的作用域、链接和存储期。存储类别由声明变量的位置和与之关联的关键字决定。定义在所有函数外部的变量具有文件作用域、外部链接、静态存储期。声明在函数中的变量是自动变量，除非该变量前面使用了其他关键字。它们具有块作用域、无链接、自动存储期。以 static 关键字声明在函数中的变量具有块作用域、无链接、静态存储期。以 static 关键字声明在函数外部的变量具有文件作用域、内部链接、静态存储期。

C11 新增了一个存储类别说明符：_Thread_local。以该关键字声明的对象具有线程存储期，意思是在线程中声明的对象在该线程运行期间一直存在，且在线程开始时被初始化。因此，这种对象属于线程私有。

属性：

下面总结了这些存储类别的属性：

存储类别	存储期	作用域	链接	如何声明
自动	自动	块	无	在块中
寄存器	自动	块	无	在块中，使用关键字 register

续表

存储类别	存储期	作用域	链接	如何声明
静态、外部链接	静态	文件	外部	在所有函数外部
静态、内部链接	静态	文件	内部	在所有函数外部，使用关键字 static
静态、无链接	静态	块	无	在块中，使用关键字 static
线程、外部链接	线程	文件	外部	在所有块的外部，使用关键字_Thread_local
线程、内部链接	线程	文件	内部	在所有块的外部，使用关键字 static 和_Thread_local
线程、无链接	线程	块	无	在块中，使用关键字 static 和_Thread_local

注意，关键字 extern 只能用来再次声明在别处已定义过的变量。在函数外部定义变量，该变量具有外部链接属性。

除了以上介绍的存储类别，C 还提供了动态分配内存。这种内存通过调用 malloc()函数系列中的一个函数来分配。这种函数返回一个可用于访问内存的指针。调用 free()函数或结束程序可以释放动态分配的内存。任何可以访问指向该内存指针的函数均可访问这块内存。例如，一个函数可以把这个指针的值返回给另一个函数，那么另一个函数也可以访问该指针所指向的内存。

B.3.3　总结：限定符

关键字

使用下面关键字限定变量：

const、volatile、restrict

一般注释

限定符用于限制变量的使用方式。不能改变初始化以后的 const 变量。编译器不会假设 volatile 变量不被某些外部代理（如，一个硬件更新）改变。restrict 限定的指针是访问它所指向内存的唯一方式（在特定作用域中）。

属性

const int joy = 101;声明创建了变量 joy，它的值被初始化为 101。

volatile unsigned int incoming;声明创建了变量 incoming，该变量在程序中两次出现之间，其值可能会发生改变。

const int * ptr = &joy;声明创建了指针 ptr，该指针不能用来改变变量 joy 的值，但是它可以指向其他位置。

int * const ptr = &joy;声明创建了指针 ptr，不能改变该指针的值，即 ptr 只能指向 joy，但是可以用它来改变 joy 的值。

void simple (const char * s);声明表明形式参数 s 被传递给 simple()的值初始化后，simple()不能改变 s 指向的值。

void supple(int * const pi);与 void supple(int pi[const]);等价。这两个声明都表明 supple()函数不会改变形参 pi。

void interleave(int * restrict p1, int * restrict p2, int n);声明表明 p1 和 p2 是访问它们所指向内存的唯一方法，这意味着这两个块不能重叠。

B.4 参考资料 IV：表达式、语句和程序流

B.4.1 总结：表达式和语句

在 C 语言中，对表达式可以求值，通过语句可以执行某些行为。

表达式

表达式由运算符和运算对象组成。最简单的表达式是一个常量或一个不带运算符的变量，如 `22` 或 `beebop`。稍复杂些的例子是 `55 + 22` 和 `vap = 2 * (vip + (vup = 4))`。

语句

大部分语句都以分号结尾。以分号结尾的表达式都是语句，但这样的语句不一定有意义。语句分为简单语句和复合语句。**简单语句**以分号结尾，如下所示：

```
toes = 12;              // 赋值表达式语句
printf("%d\n", toes);  // 函数调用表达式语句
;                       //空语句，什么也不做
```

（注意，在 C 语言中，声明不是语句。）

用花括号括起来的一条或多条语句是**复合语句**或**块**。如下面的 `while` 语句所示：

```
while (years < 100)
{
    wisdom = wisdom + 1;
    printf("%d %d\n", years, wisdom);
    years = years + 1;
}
```

B.4.2 总结：while 语句

关键字

`while` 语句的关键字是 `while`。

一般注释

`while` 语句创建了一个循环，在 *expression* 为假之前重复执行。`while` 语句是一个**入口条件**循环，在下一轮迭代之前先确定是否要再次循环。因此可能一次循环也不执行。*statement* 可以是一个简单语句或复合语句。

形式

```
while ( expression )
statement
```

当 *expression* 为假（或 `0`）之前，重复执行 *statement* 部分。

示例

```
while (n++ < 100)
    printf(" %d %d\n",n, 2*n+1);

while (fargo < 1000)
{
    fargo = fargo + step;
    step = 2 * step;
}
```

B.4.3　总结：for 语句

关键字

for 语句的关键字是 for。

一般注释

for 语句使用 3 个控制表达式控制循环过程，分别用分号隔开。*initialize* 表达式在执行 for 语句之前只执行一次；然后对 *test* 表达式求值，如果表达式为真（或非零），执行循环一次；接着对 *update* 表达式求值，并再次检查 *test* 表达式。for 语句是一种入口条件循环，即在执行循环之前就决定了是否执行循环。因此，for 循环可能一次都不执行。*statement* 部分可以是一条简单语句或复合语句。

形式：

```
for ( initialize; test; update )
statement
```

在 *test* 为假或 0 之前，重复执行 *statement* 部分。

C99 允许在 for 循环头中包含声明。变量的作用域和生命期被限制在 for 循环中。

示例：

```
for (n = 0; n < 10 ; n++)
    printf(" %d %d\n", n, 2 * n + 1);
for (int k = 0; k < 10 ; ++k)  // C99
  printf("%d %d\n", k, 2 * k+1);
```

B.4.4　总结：do　while 语句

关键字

do　while 语句的关键字是 do 和 while。

一般注解：

do　while 语句创建一个循环，在 *expression* 为假或 0 之前重复执行循环体中的内容。do　while 语句是一种出口条件循环，即在执行完循环体后才根据测试条件决定是否再次执行循环。因此，该循环至少必须执行一次。*statement* 部分可是一条简单语句或复合语句。

形式：

```
do
statement
while ( expression );
```

在 *test* 为假或 0 之前，重复执行 *statement* 部分。

示例：

```
do
    scanf("%d", &number);
while (number != 20);
```

B.4.5　总结：if 语句

小结：用 if 语句进行选择

关键字：if、else

一般注解：

下面各形式中，*statement* 可以是一条简单语句或复合语句。表达式为真说明其值是非零值。

形式 1：

```
if (expression)
statement
```

如果 *expression* 为真，则执行 *statement* 部分。

形式 2：

```
if (expression)
statement1
else
statement2
```

如果 *expression* 为真，执行 *statement1* 部分；否则，执行 *statement2* 部分。

形式 3：

```
if (expression1)
statement1
else if (expression2)
statement2
else
statement3
```

如果 *expression1* 为真，执行 *statement1* 部分；如果 *expression2* 为真，执行 *statement2* 部分；否则，执行 *statement3* 部分。

示例：

```
if (legs == 4)
    printf("It might be a horse.\n");
else if (legs > 4)
    printf("It is not a horse.\n");
else     /* 如果 legs < 4 */
{
    legs++;
printf("Now it has one more leg.\n");
}
```

B.4.6　带多重选择的 switch 语句

关键字：switch

一般注解：

程序控制根据 *expression* 的值跳转至相应的 case 标签处。然后，程序流执行剩下的所有语句，除非执行到 break 语句进行重定向。*expression* 和 case 标签都必须是整数值（包括 char 类型），标签必须是常量或完全由常量组成的表达式。如果没有 case 标签与 *expression* 的值匹配，控制则转至标有 default 的语句（如果有的话）；否则，控制将转至紧跟在 switch 语句后面的语句。控制转至特定标签后，将执行 switch 语句中其后的所有语句，除非到达 switch 末尾，或执行到 break 语句。

形式：

```
switch ( expression )
{
    case label1 : statement1//使用 break 跳出 switch
    case label2 : statement2
    default     : statement3
}
```

可以有多个标签语句，default 语句可选。

示例：

```
switch (value)
{
    case 1 : find_sum(ar, n);
             break;
    case 2 : show_array(ar, n);
             break;
    case 3 : puts("Goodbye!");
             break;
    default : puts("Invalid choice, try again.");
              break;
}

switch (letter)
{
    case 'a' :
    case 'e' : printf("%d is a vowel\n", letter);
    case 'c' :
    case 'n' : printf("%d is in \"cane\"\n", letter);
    default : printf("Have a nice day.\n");
}
```

如果 letter 的值是'a'或'e'，就打印这 3 条消息；如果 letter 的值是'c'或'n'，则只打印后两条消息；letter 是其他值时，值打印最后一条消息。

B.4.7　总结：程序跳转

关键字：break、continue、goto

一般注解：

这 3 种语句都能使程序流从程序的一处跳转至另一处。

break 语句：

所有的循环和 switch 语句都可以使用 break 语句。它使程序控制跳出当前循环或 switch 语句的剩余部分，并继续执行跟在循环或 switch 后面的语句。

示例：

```
while ((ch = getchar()) != EOF)
{
    putchar(ch);
    if (ch == ' ')
        break;        // 结束循环
    chcount++;
}
```

continue 语句：

所有的循环都可以使用 continue 语句，但是 switch 语句不行。continue 语句使程序控制跳出循环的剩余部分。对于 while 或 for 循环，程序执行到 continue 语句后会开始进入下一轮迭代。对于 do while 循环，对出口条件求值后，如有必要会进入下一轮迭代。

示例：

```
while ((ch = getchar()) != EOF)
{
    if (ch == ' ')
        continue; // 跳转至测试条件
    putchar(ch);
```

```
    chcount++;
}
```

以上程序段打印用户输入的内容并统计非空格字符

goto 语句：

goto 语句使程序控制跳转至相应标签语句。冒号用于分隔标签和标签语句。标签名遵循变量命名规则。标签语句可以出现在 goto 的前面或后面。

形式：

```
goto label ;
    .
    .
    .
label : statement
```

示例：

```
top : ch = getchar();
    .
    .
    .
    if (ch != 'y')
    goto top;
```

B.5　参考资料 V：新增 C99 和 C11 的 ANSI C 库

ANSI C 库把函数分成不同的组，每个组都有相关联的头文件。本节将概括地介绍库函数，列出头文件并简要描述相关的函数。文中会较详细地介绍某些函数（例如，一些 I/O 函数）。欲了解完整的函数说明，请参考具体实现的文档或参考手册，或者试试这个在线参考：http://www.acm.uiuc .edu/webmonkeys/book/c_guide/。

B.5.1　断言：assert.h

assert.h 头文件中把 assert()定义为一个宏。在包含 assert.h 头文件之前定义宏标识符 NDEBUG，可以禁用 assert()宏。通常用一个关系表达式或逻辑表达式作为 assert()的参数，如果运行正常，那么程序在执行到该点时，作为参数的表达式应该为真。表 B.5.1 描述了 assert()宏。

表 B.5.1　断　言　宏

原型	描述
void assert(int exprs);	如果 exprs 为 0（或真），宏什么也不做。如果 exprs 为 0（或假），assert()就显示该表达式和其所在的行号和文件名。然后，assert()调用 abort()

C11 新增了 static_assert 宏，展开为_Static_assert。_Static_assert 是一个关键字，被认为是一种声明形式。它以这种方式提供一个编译时检查：

_Static_assert(常量表达式,字符串字面量);

如果对常量表达式求值为 0，编译器会给出一条包含字符串字面量的错误消息；否则，没有任何效果。

B.5.2　复数：complex.h（C99）

C99 标准支持复数计算，C11 进一步支持了这个功能。实现除提供_Complex 类型外还可以选择是否提供_Imaginary 类型。在 C11 中，可以选择是否提供这两种类型。C99 规定，实现必须提供_Complex 类型，但是_Imaginary 类型为可选，可以提供或不提供。附录 B 的参考资料 VIII 中进一步讨论了 C 如何

支持复数。complex.h 头文件中定义了表 B.5.2 所列的宏。

表 B.5.2　complex.h 宏

宏	描述
complex	展开为类型关键字_Complex
_Complex_I	展开为 const float _Complex 类型的表达式，其值的平方是-1
imaginary	如果支持虚数类型，展开为类型关键字_Imaginary
_Imaginary_I	如果支持虚数类型，展开为 const float _Imaginary 类型的表达式，其值的平方是-1
I	展开为_Complex_I 或_Imaginary_I

对于实现复数方面，C 和 C++不同。C 通过 complex.h 头文件支持，而 C++通过 complex 头文件支持。而且，C++使用类来定义复数类型。

可以使用 STDC CX_LIMITED_RANGE 编译指令来表明是使用普通的数学公式（设置为 on 时），还是要特别注意极值（设置为 off 时）：

```
#include <complex.h>
#pragma STDC CX_LIMITED_RANGE on
```

库函数分为 3 种：double、float、long double。表 B.5.3 列出了 double 版本的函数。float 和 long double 版本只需要在函数名后面分别加上 f 和 l。即 csinf()就是 csin()的 float 版本，而 csinl()是 csin()的 long double 版本。另外要注意，角度的单位是弧度。

表 B.5.3　复 数 函 数

原型	描述
double complex cacos(double complex z);	返回 z 的复数反余弦
double complex casin(double complex z);	返回 z 的复数反正弦
double complex catan(double complex z);	返回 z 的复数反正切
double complex ccos(double complex z);	返回 z 的复数余弦
double complex csin(double complex z);	返回 z 的复数正弦
double complex ctan(double complex z);	返回 z 的复数正切
double complex cacosh(double complex z);	返回 z 的复数反双曲余弦
double complex casinh(double complex z);	返回 z 的复数反双曲正弦
double complex catanh(double complex z);	返回 z 的复数反双曲正切
double complex ccosh(double complex z);	返回 z 的复数双曲余弦
double complex csinh(double complex z);	返回 z 的复数双曲正弦
double complex ctanh(double complex z);	返回 z 的复数双曲正切
double complex cexp(double complex z);	返回 e 的 z 次幂复数值
double complex clog(double complex z);	返回 z 的自然对数（以 e 为底）的复数值
double cabs(double complex z);	返回 z 的绝对值（或大小）
double complex cpows(double complex z, double complex y);	返回 z 的 y 次幂
double complex csqrt(double complex z);	返回 z 的复数平方根

续表

原型	描述
double carg(double complex z);	以弧度为单位返回 z 的相位角（或幅角）
double cimag(double complex z);	以实数形式返回 z 的虚部
double complex conj(double complex z);	返回 z 的共轭复数
double complex cproj(double complex z);	返回 z 在黎曼球面上的投影
double complex CMPLX(double x,double y);	返回实部为 x、虚部为 y 的复数（C11）
double creal(double complex z);	以实数形式返回 z 的实部

B.5.3　字符处理：ctype.h

这些函数都接受 int 类型的参数，这些参数可以表示为 unsigned char 类型的值或 EOF。使用其他值的效果是未定义的。在表 B.5.4 中，“真”表示“非 0 值”。对一些定义的解释取决于当前的本地设置，这些由 locale.h 中的函数来控制。该表显示了在解释本地化的“C”时要用到的一些函数。

表 B.5.4　字符处理函数

原型	描述
int isalnum(int c);	如果 c 是字母或数字，则返回真
int isalpha(int c);	如果 c 是字母，则返回真
int isblank(int c);	如果 c 是空格或水平制表符，则返回真（C99）
int iscntrl(int c);	如果 c 是控制字符（如 Ctrl+B），则返回真
int isdigit(int c);	如果 c 是数字，则返回真
int isgraph(int c);	如果 c 是非空格打印字符，则返回真
int islower(int c);	如果 c 是小写字符，则返回真
int isprint(int c);	如果 c 是打印字符，则返回真
int ispunct(int c);	如果 c 是标点字符（除了空格、字母、数字以外的字符），则返回真
int isspace(int c);	如果 c 是空白字符（空格、换行符、换页符、回车符、垂直或水平制表符，或者其他实现定义的字符），则返回真
int isupper(int c);	如果 c 是大写字符，则返回真
int isxdigit(int c);	如果 c 是十六进制数字字符，则返回真
int tolower(int c);	如果 c 是大写字符，则返回其小写字符；否则返回 c
int toupper(int c);	如果 c 是小写字符，则返回其大写字符；否则返回 c

B.5.4　错误报告：errno.h

errno.h 头文件支持较老式的错误报告机制。该机制提供一个标识符（或有时称为宏）ERRNO 可访问的外部静态内存位置。一些库函数把一个值放进这个位置用于报告错误，然后包含该头文件的程序就可以通过查看 ERRNO 的值检查是否报告了一个特定的错误。ERRNO 机制被认为不够艺术，而且设置 ERRNO 值也不需要数学函数了。标准提供了 3 个宏值表示特殊的错误，但是有些实现会提供更多。表 B.5.5 列出了这些标准宏。

表 B.5.5　errno.h 宏

宏	含义
EDOM	函数调用中的域错误（参数越界）
ERANGE	函数返回值的范围错误（返回值越界）
EILSEQ	宽字符转换错误

B.5.5　浮点环境：fenv.h（C99）

C99 标准通过 fenv.h 头文件提供访问和控制浮点环境。

浮点环境（*floating-point environment*）由一组**状态标志**（*status flag*）和**控制模式**（*control mode*）组成。在浮点计算中发生异常情况时（如，被零除），可以“抛出一个异常”。这意味着该异常情况设置了一个浮点环境标志。控制模式值可以进行一些控制，例如控制舍入的方向。fenv.h 头文件定义了一组宏表示多种异常情况和控制模式，并提供了与环境交互的函数原型。头文件还提供了一个编译指令来启用或禁用访问浮点环境的功能。

下面的指令开启访问浮点环境：

```
#pragma STDC FENV_ACCESS on
```

下面的指令关闭访问浮点环境：

```
#pragma STDC FENV_ACCESS off
```

应该把该编译指示放在所有外部声明之前或者复合块的开始处。在遇到下一个编译指示之前、或到达文件末尾（外部指令）、或到达复合语句的末尾（块指令），当前编译指示一直有效。

头文件定义了两种类型，如表 B.5.6 所示。

表 B.5.6　fenv.h 类型

类型	表示
fenv_t	整个浮点环境
fexcept_t	浮点状态标志集合

头文件定义了一些宏，表示一些可能发生的浮点异常情况控制状态。其他实现可能定义更多的宏，但是必须以 FE_开头，后面跟大写字母。表 B.5.7 列出了一些标准异常宏。

表 B.5.7　fenv.h 中的标准异常宏

宏	含义
FE_DIVBYZERO	抛出被零除异常
FE_INEXACT	抛出不精确值异常
FE_INVALID	抛出无效值异常
FE_OVERFLOW	抛出上溢异常
FE_UNDERFLOW	抛出下溢异常
FE_ALL_EXCEPT	实现支持的所有浮点异常的按位或
FE_DOWNWARD	向下舍入
FE_TONEAREST	向最近的舍入
FE_TOWARDZERO	趋 0 舍入
FE_UPWARD	向上舍入
FE_DFL_ENV	表示默认环境，类型是 const fenv_t *

表 B.5.8 中列出了 fenv.h 头文件中的标准函数原型。注意，常用的参数值和返回值与表 B.5.7 中的宏相对应。例如，FE_UPWARD 是 fesetround()的一个合适参数。

表 B.5.8　fenv.h 中的标准函数原型

原型	描述
void feclearexcept(int excepts);	清理 excepts 表示的异常
void fegetexceptflag(fexcept_t *flagp, int excepts);	把 excepts 指明的浮点状态标志储存在 flagp 指向的对象中
void feraiseexcept(int excepts);	抛出 excepts 指定的异常
void fesetexceptflag(const fexcept_t *flagp, intexcepts);	把 excepts 指明的浮点状态标志设置为 flagp 的值；在此之前，fegetexceptflag()调用应该设置 flagp 的值
int fetestexcept(int excepts);	测试 excepts 指定的状态标志；该函数返回指定状态标志的按位或
int fegetround(void);	返回当前的舍入方向
int fesetround(int round);	把舍入方向设置为 round 的值；当且仅当设置成功时，函数返回 0
void fegetenv(fenv_t *envp);	把当前环境储存至 envp 指向的位置中
int feholdexcept(fenv_t *envp);	把当前浮点环境储存至 envp 指向的位置中，清除浮点状态标志，然后如果可能的话就设置非停模式（*nonstop mode*），在这种模式中即使发生异常也继续执行。当且仅当执行成功时，函数返回 0
void fesetenv(const fenv_t *envp);	建立 envp 表示的浮点环境；envp 应指向一个之前通过调用 fegetenv()、feholdexcept()或浮点环境宏设置的数据对象
void feupdateenv (const fenv_t *envp);	函数在自动存储区中储存当前抛出的异常，建立 envp 指向的对象表示的浮点环境，然后抛出已储存的浮点异常；envp 应指向一个之前通过调用 fegetenv()、feholdexcept()或浮点环境宏设置的数据对象

B.5.6　浮点特性：float.h

float.h 头文件中定义了一些表示各种限制和形参的宏。表 B.5.9 列出了这些宏，C11 新增的宏以斜体并缩进标出。许多宏都涉及下面的浮点表示模型：

$$x = sb^e \sum_{k=1}^{p} f_k^{b^{-k}}$$

如果第 1 个数 f_1 是非 0（且 x 是非 0），该数字被称为**标准化浮点数**。附录 B 的参考资料 VIII 中将更详细地解释一些宏。

表 B.5.9　float.h 宏

宏	含义
FLT_ROUNDS	默认舍入方案
FLT_EVAL_METHOD	浮点表达式求值的默认方案
FLT_HAS_SUBNORM	存在或缺少 float 类型的反常值
DBL_HAS_SUBNORM	存在或缺少 double 类型的反常值
LDBL_HAS_SUBNORM	存在或缺少 long double 类型的反常值
FLT_RADIX[1]	指数表示法中使用的进制数(b)，最小值为 2

[1] FLT_RADIX 用于表示 3 种浮点数类型的基数。——译者注

续表

宏	含义
FLT_MANT_DIG	以 FLT_RADIX 进制表示的 float 类型数的位数（模型中的 p）
DBL_MANT_DIG	以 FLT_RADIX 进制表示的 double 类型数的位数（模型中的 p）
LDBL_MANT_DIG	以 FLT_RADIX 进制表示的 long double 类型数的位数（模型中的 p）
FLT_DECIMAL_DIG	在 b 进制和十进制相互转换不损失精度的前提下，float 类型的十进制数的位数（最小值是 6）
DBL_DECIMAL_DIG	在 b 进制和十进制相互转换不损失精度的前提下，double 类型的十进制数的位数（最小值是 10）
LDBL_DECIMAL_DIG	在 b 进制和十进制相互转换不损失精度的前提下，long double 类型的十进制数的位数（最小值是 10）
DECIMAL_DIG	在 b 进制与十进制相互转换不损失精度的前提下，浮点类型十进制数的最大个数（最小值为 10）
FLT_DIG	在不损失精度的前提下，float 类型可表示的十进制数位数（最小值为 6）
DBL_DIG	在不损失精度的前提下，double 类型可表示的十进制数位数（最小值为 10）
LDBL_DIG	在不损失精度的前提下，long double 类型可表示的十进制数位数（最小值为 10）
FLT_MIN_EXP	float 类型 e 表示法，指数的最小负正整数值
DBL_MIN_EXP	double 类型 e 表示法，指数的最小负正整数值
LDBL_MIN_EXP	long double 类型 e 表示法，指数的最小负正整数值
FLT_MIN_10_EXP	用 10 的 x 次幂表示规范化 float 类型数时，x 的最小负整数值（不超过-37）
DBL_MIN_10_EXP	用 10 的 x 次幂表示规范化 double 类型数时，x 的最小负整数值（不超过-37）
LDBL_MIN_10_EXP	用 10 的 x 次幂表示规范化 long double 类型数时，x 的最小负整数值（不超过-37）
FLT_MAX_EXP	float 类型 e 表示法，指数的最大正整数值
DBL_MAX_EXP	double 类型 e 表示法，指数的最大正整数值
LDBL_MAX_EXP	long double 类型 e 表示法，指数的最大正整数值
FLT_MAX_10_EXP	用 10 的 x 次幂表示规范化 float 类型数时，x 的最大正整数值（至少+37）
DBL_MAX_10_EXP	用 10 的 x 次幂表示规范化 double 类型数时，x 的最大正整数值（至少+37）
LDBL_MAX_10_EXP	用 10 的 x 次幂表示规范化 long double 类型数时，x 的最大正整数值（至少+37）
FLT_MAX	float 类型的最大有限值（至少 1E+37）
DBL_MAX	doble 类型的最大有限值（至少 1E+37）
LDBL_MAX	long double 类型的最大有限值（至少 1E+37）
FLT_EPSILON	float 类型比 1 大的最小值与 1 的差值（不超过 1E-9）
DBL_EPSILON	double 类型比 1 大的最小值与 1 的差值（不超过 1E-9）
LDBL_EPSILON	long double 类型比 1 大的最小值与 1 的差值（不超过 1E-9）
FLT_MIN	标准化 float 类型的最小正值（不超过 1E-37）
DBL_MIN	标准化 double 类型的最小正值（不超过 1E-37）
LDBL_MIN	标准化 long double 类型的最小正值（不超过 1E-37）
FLT_TRUE_MIN	float 类型的最小正值（不超过 1E-37）
DBL_TRUE_MIN	double 类型的最小正值（不超过 1E-37）
LDBL_TRUE_MIN	long double 类型的最小正值（不超过 1E-37）

B.5.7　整数类型的格式转换：inttypes.h

该头文件定义了一些宏可用作转换说明来扩展整数类型。参考资料 VI“扩展的整数类型”将进一步讨论。该头文件还声明了这个类型：imaxdiv_t。这是一个结构类型，表示 idivmax()函数的返回值。

该头文件中还包含 stdint.h，并声明了一些使用最大长度整数类型的函数，这种整数类型在 stdint.h 中声明为 intmax。表 B.5.10 列出了这些函数。

表 B.5.10　使用最大长度整数的函数

原型	描述
intmax_t imaxabs(intmax_t j);	返回 j 的绝对值
imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);	单独计算 numer/denom 的商和余数，并把两个计算结果储存在返回的结构中
intmax_t strtoimax(const char * restrict nptr, char ** restrict endptr, int base);	相当于 strtol()函数，但是该函数把字符串转换成 intmax_t 类型并返回该值
uintmax_t strtoumax(const char * restrict nptr, char ** restrict endptr, int base);	相当于 strtoul()函数，但是该函数但是该函数把字符串转换成 intmax_t 类型并返回该值
intmax_t wcstoimax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);	strtoimax()函数的 wchar_t 类型的版本
uintmax_t wcstoumax(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);	strtoumax()函数的 wchar_t 类型的版本

B.5.8　可选拼写：iso646.h

该头文件提供了 11 个宏，扩展了指定的运算符，如表 B.5.11 所列。

表 B.5.11　可 选 拼写

宏	运算符	宏	运算符	宏	运算符
and	&&	and_eq	&=	bitand	&
bitor	|	compl	～	not	!
not	!=	or	||	or_eq	|=
xor	^	xor_eq	^=		

B.5.9　本地化：locale.h

本地化是一组设置，用于控制一些特定的设置项，如表示小数点的符号。本地值储存在 struct lconv 类型的结构中，定义在 locale.h 头文件中。可以用一个字符串来指定本地化，该字符串指定了一组结构成员的特殊值。默认的本地化由字符串"C"指定。表 B.5.12 列出了本地化函数，后面做了简要说明。

表 B.5.12　本地化函数

原型	描述
char * setlocale(int category, const char * locale);	该函数把某些值设置为本地和 locale 指定的值。category 的值决定要设置哪些本地值（参见 B.5.13）。如果成功设置本地化，该函数将返回一个在新本地化中与指定类别相关联的指针；如果不能完成本地化请求，则返回空指针
struct lconv *localeconv(void);	返回一个指向 struct lconv 类型结构的指针，该结构中储存着当前的本地值

setlocale()函数的 locale 形参所需的值可能是默认值"C"，也可能是""，表示实现定义的本地环境。实现可以定义更多的本地化设置。category 形参的值可能由表 B.5.13 中所列的宏表示。

表 B.5.13　category 宏

原型	描述
NULL	本地化设置不变，返回指向当前本地化的指针
LC_ALL	改变所有的本地值
LC_COLLATE	改变 strcoll()和 strxfrm()所用的排列顺序的本地值
LC_CTYPE	改变字符处理函数和多字节函数的本地值
LC_MONETARY	改变货币格式信息的本地值
LC_NUMERIC	改变十进制小数点符号和格式化 I/O 使用的非货币格式本地值
LC_TIME	改变 strftime()所用的时间格式本地值

表 B.5.14 列出了 struct lconv 结构所需的成员。

表 B.5.14　struct lcconv 所需的成员

成员变量	描述
char *decimal_point	非货币值的小数点字符
char *thousands_sep	非货币值中小数点前面的千位分隔符
char *grouping	一个字符串，表示非货币量中每组数字的大小
char *int_curr_symbol	国际货币符号
char *currency_symbol	本地货币符号
char *mon_decimal_point	货币值的小数点符号
char *mon_thousands_sep	货币值的千位分隔符
char *mon_grouping	一个字符串，表示货币量中每组数字的大小
char *positive_sign	指明非负格式化货币值的字符串
char *negative_sign	指明负格式化货币值的字符串
char int_frac_digits	国际格式化货币值中，小数点后面的数字个数
char frac_digits	本地格式化货币值中，小数点后面的数字个数
char p_cs_precedes	如果该值为 1，则 currency_symbol 在非负格式化货币值的前面； 如果该值为 0，则 currency_symbol 在非负格式化货币值的后面

续表

成员变量	描述
char p_sep_by_space	如果该值为 1，则用空格把 currency_symbol 和非负格式化货币值隔开； 如果该值为 0，则不用空格分隔 currency_symbol 和非负格式化货币值
char n_cs_precedes	如果该值为 1，则 currency_symbol 在负格式化货币值的前面； 如果该值为 0，则 currency_symbol 在负格式化货币值的后面
char n_sep_by_space	如果该值为 1，则用空格把 currency_symbol 和负格式化货币值隔开； 如果该值为 0，则不用空格分隔 currency_symbol 和负格式化货币值
char p_sign_posn	其值表示 positive_sign 字符串的位置： 0 表示用圆括号把数值和货币符号括起来 1 表示字符串在数值和货币符号前面 2 表示字符串在数值和货币符号后面 3 表示直接把字符串放在货币前面 4 表示字符串紧跟在货币符号后面
char n_sign_posn	其值表示 negative_sign 字符串的位置，含义与 p_sign_posn 相同
char int_p_cs_precedes	如果该值为 1，则 int_currency_symbol 在非负格式化货币值的前面； 如果该值为 0，则 int_currency_symbol 在非负格式化货币值的后面
char int_p_sep_by_space	如果该值为 1，则用空格把 int_currency_symbol 和非负格式化货币值隔开； 如果该值为 0，则不用空格分隔 int_currency_symbol 和非负格式化货币值
char int_n_cs_precedes	如果该值为 1，则 int_currency_symbol 在负格式化货币值的前面； 如果该值为 0，则 int_currency_symbol 在负格式化货币值的后面
char int_n_sep_by_space	如果该值为 1，则用空格把 int_currency_symbol 和负格式化货币值隔开； 如果该值为 0，则不用空格分隔 int_currency_symbol 和负格式化货币值
char int_p_sign_posn	其值表示 positive_sign 相对于非负国际格式化货币值的位置
char int_n_sign_posn	其值表示 negative_sign 相对于负国际格式化货币值的位置

B.5.10　数学库：math.h

C99 为 math.h 头文件定义了两种类型：float_t 和 double_t。这两种类型分别与 float 和 double 类型至少等宽，是计算 float 和 double 时效率最高的类型。

该头文件还定义了一些宏，如表 B.5.15 所列。该表中除了 HUGE_VAL 外，都是 C99 新增的。在参考资料 VIII：“C99 数值计算增强”中会进一步详细介绍。

表 B.5.15　math.h 宏

宏	描述
HUGE_VAL	正双精度常量，不一定能用浮点数表示；在过去，函数的计算结果超过了可表示的最大值时，就用它作为函数的返回值
HUGE_VALF	与 HUGE_VAL 类似，适用于 float 类型
HUGE_VALL	与 HUGE_VAL 类似，适用于 long double 类型
INFINITY	如果允许的话，展开为一个表示无符号或正无穷大的常量 float 表达式；否则，展开为一个在编译时溢出的正浮点常量

续表

宏	描述
NAN	当且仅当实现支持 float 类型的 NaN 时才被定义(NaN 是 Not-a-Number 的缩写，表示"非数"，用于处理计算中的错误情况，如除以 0.0 或求负数的平方根)
FP_INFINITE	分类数，表示一个无穷大的浮点值
FP_NAN	分类数，表示一个不是数的浮点值
FP_NORMAL	分类数，表示一个正常的浮点值
FP_SUBNORMAL	分类数，表示一个低于正常浮点值的值（精度被降低）
FP_ZERO	分类数，表示 0 的浮点值
FP_FAST_FMA	（可选）如果已定义，对于 double 类型的运算对象，该宏表明 fma()函数与先乘法运算后加法运算的速度相当或更快
FP_FAST_FMAF	（可选）如果已定义，对于 double 类型的运算对象，该宏表明 fmaf()函数与先乘法运算后加法运算的速度相当或更快
FP_FAST_FMAL	（可选）如果已定义，对于 long double 类型的运算对象，该宏表明 fmal()函数与先乘法运算后加法运算的速度相当或更快
FP_ILOGB0	整型常量表达式，表示 ilogn(0)的返回值
FP_ILOGBNAN	整型常量表达式，表示 ilogn(NaN)的返回值
MATH_ERRNO	展开为整型常量 1
MATH_ERREXCEPT	展开为整型常量 2
math_errhandling	值为 MATH_ERRNO、MATH_ERREXCEPT 或这两个值的按位或

数学函数通常使用 double 类型的值。C99 新增了这些函数的 float 和 long double 版本，其函数名为分别在原函数名后添加 f 后缀和 l 后缀。例如，C 语言现在提供这些函数原型：

```
double sin(double);
float sinf(float);
long double sinl(long double);
```

篇幅有限，表 B.5.16 仅列出了数学库中这些函数的 double 版本。该表引用了 FLT_RADIX，该常量定义在 float.h 中，代表内部浮点表示法中幂的底数。最常用的值是 2。

表 B.5.16　ANSI C 标准数学函数

原型	描述
int classify(*real-floating* x);	C99 宏，返回适合 x 的浮点分类值
int isfinite(*real-floating* x);	C99 宏，当且仅当 x 为有穷时返回一个非 0 值
int isfin(*real-floating*x);	C99 宏，当且仅当 x 为无穷时返回一个非 0 值
int isnan(*real-floating*x);	C99 宏，当且仅当 x 为 NaN 时返回一个非 0 值
int isnormal(*real-floating*x);	C99 宏，当且仅当 x 为正常数时返回一个非 0 值
int signbit(*real-floating* x);	C99 宏，当且仅当 x 的符号为负时返回一个非 0 值
double acos(double x);	返回余弦为 x 的角度（0～π 弧度）
double asin(double x);	返回正弦为 x 的角度（-π/2～π/2 弧度）
double atan(double x);	返回正切为 x 的角度（-π/2～π/2 弧度）

续表

原型	描述
double atan2(double y, double x);	返回正切为 y/x 的角度（-π～π弧度）
double cos(double x);	返回 x（弧度）的余弦值
double sin(double x);	返回 x（弧度）的正弦值
double tan(double x);	返回 x（弧度）的正切值
double cosh(double x);	返回 x 的双曲余弦值
double sinh(double x);	返回 x 的双曲正弦值
double tanh(double x);	返回 x 的双曲切值
double exp(double x);	返回 e 的 x 次幂（e^x）
double exp2(double x);	返回 2 的 x 次幂（2^x）
double expm1(double x);	返回 e^x - 1（C99）
double frexp(double v, int *pt_e);	把 v 的值分成两部分，一个是返回的规范化小数；一个是 2 的幂，储存在 pt_e 指向的位置上
int ilogb(double x);	以 signed int 类型返回 x 的指数（C99）
double ldexp(double x, int p);	返回 x 乘以 2 的 p 次幂（即 x * 2^p）
double log(double x);	返回 x 的自然对数
double log10(double x);	返回以 10 为底 x 的对数
double log1p(double x);	返回 log(1 + x)（C99）
double log2(double x);	返回以 2 为底 x 的对数（C99）
double logb(double x);	返回 FLT_RADIX（系统内部浮点表示法中幂的底数）为底 x 的有符号对数（C99）
double modf(double x, double *p);	把 x 分成整数部分和小数部分，两部分的符号相同，返回小数部分，并把整数部分储存在 p 所指向的位置上
double scalbn(double x, int n);	返回 x × FLT_RADIX^n（C99）
double scalbln(double x, long n);	返回 x × FLT_RADIX^n（C99）
double cbrt(double x);	返回 x 的立方根（C99）
double hypot(double x, double y);	返回 x 平方与 y 平方之和的平方根（C99）
double pow(double x, double y);	返回 x 的 y 次幂
double sqrt(double x);	返回 x 的平方根
double erf(double x);	返回 x 的误差函数（C99）
double lgamma(double x);	返回 x 的伽马函数绝对值的自然对数（C99）
double tgamma(double x);	返回 x 的伽马函数（C99）
double ceil(double x);	返回不小于 x 的最小整数值
double fabs(double x);	返回 x 的绝对值
double floor(double x);	返回不大于 x 的最大值
double nearbyint(double x);	以浮点格式把 x 四舍五入为最接近的整数；使用浮点环境指定的舍入规则（C99）

续表

原型	描述
double rint(double x);	与 nearbyint()类似，但是该函数会抛出“不精确”异常
long int lrint(double x);	以 long int 格式把 x 舍入为最接近的整数；使用浮点环境指定的舍入规则（C99）
long long int llrint(double x);	以 long long int 格式把 x 舍入为最接近的整数；使用浮点环境指定的舍入规则（C99）
double round(double x);	以浮点格式把 x 舍入为最接近的整数，总是四舍五入
long int lround(double x);	与 round()类似，但是该函数返回值的类型是 long int
long long int llround(double x);	与 round()类似，但是该函数返回值的类型是 long long int
double trunc(double x);	以浮点格式把 x 舍入为最接近的整数，其结果的绝对值不大于 x 的绝对值（C99）
int fmod(double x, double y);	返回 x/y 的小数部分，如果 y 不是 0，则其计算结果的符号与 x 相同，而且该结果的绝对值要小于 y 的绝对值
double remainder(double x, double y);	返回 x 除以 y 的余数，IEC 60559 定义为 x - n*y，n 取与 x/y 最接近的整数；如果(n - x/y)的绝对值是 1/2，n 取偶数
double remquo(double x, double y, int *quo);	返回与 remainder()相同的值；把 x/y 的整数大小求模 2^k 的值储存在 quo 所指向的位置中，符号与 x/y 的符号相同，其中 k 为整数，至少是 3，具体值因实现而异（C99）
double copysign(double x, double y);	返回 x 的大小和 y 的符号（C99）
double nan(const char *tagp);	返回以double类型表示的quiet NaN[1]； nan("n-char-seq") 与 strtod("NAN(n-char-seq)", (char **)NULL) 等价；nan("") 与 strtod("NAN()", (char**)NULL)等价。如果不支持 quiet NaN，则返回 0
double nextafter(double x, double y);	返回 x 在 y 方向上可表示的最接近的 double 类型值；如果 x 等于 y，则返回 x（C99）
double nexttoward(double x, long double y);	与 nextafter()类似，但该函数的第 2 个参数是 long double 类型；如果 x 等于 y，则返回转换为 double 类型的 y（C99）
double fdim(double x, double y);	如果 x 大于 y，则返回 x - y 的值；如果 x 小于或等于 y，则返回 0（C99）
double fmax(double x, double y);	返回参数的最大值，如果一个参数是 NaN、另一个参数是数值，则返回数值（C99）
double fmin(double x, double y);	返回参数的最小值，如果一个参数是 NaN、另一个参数是数值，则返回数值（C99）
double fma(double x, double y, double z);	返回三元运算(x*y)+z 的大小，只在最后舍入一次（C99）
int isgreater(real-floating x, real-floating y);	C99 宏，返回(x)>(y)的值，如果有参数是 NaN，不会抛出“无效”浮点异常
int isgreaterequal(real-floating x,real-floating y);	C99 宏，返回(x)>=(y)的值，如果有参数是 NaN，不会抛出无效浮点异常

[1] NaN 分为两类：quite NaN 和 singaling NaN。两者的区别是：quite NaN 的尾数部分最高位定义为 1，而 singaling NaN 最高位定义为 0。——译者注

续表

原型	描述
intisless(real-floating x, real-floatingy);	C99 宏，返回(x)<(y)的值，如果有参数是 NaN，不会抛出无效浮点异常
int islessequal(real-floating x, real-floating y);	C99 宏，返回(x)<=(y)的值，如果有参数是 NaN，不会抛出无效浮点异常
int islessgreater(real-floating x, real-floating y);	C99 宏，返回(x)<(y)|| (x)>(y)的值，如果有参数是 NaN，不会抛出无效浮点异常
int isunordered(real-floating x, real-floating y);	如果参数不按顺序排列（至少有一个参数是 NaN），函数返回 1；否则，返回 0

B.5.11　非本地跳转：setjmp.h

setjmp.h 头文件可以让你不遵循通常的函数调用、函数返回顺序。setjmp()函数把当前执行环境的信息（例如，指向当前指令的指针）储存在 jmp_buf 类型（定义在 setjmp.h 头文件中的数组类型）的变量中，然后 longjmp()函数把执行转至这个环境中。这些函数主要是用来处理错误条件，并不是通常程序流控制的一部分。表 B.5.17 列出了这些函数。

表 B.5.17　setjmp.h 中的函数

原型	描述
int setjmp(jmp_buf env);	把调用环境储存在数组 env 中，如果是直接调用，则返回 0；如果是通过 longjmp()调用，则返回非 0
void longjmp(jmp_buf env, int val);	恢复最近的 setjmp()调用（设置 env 数组）储存的环境；完成后，程序继续像调用 setjmp()那样执行该函数，返回 val（但是该函数不允许返回 0，会将其转换成 1）

B.5.12　信号处理：signal.h

信号（*signal*）是在程序执行期间可以报告的一种情况，可以用正整数表示。raise()函数发送（或抛出）一个信号，signal()函数设置特定信号的响应。

标准定义了一个整数类型：sig_atomic_t，专门用于在处理信号时指定原子对象。也就是说，更新原子类型是不可分割的过程。

标准提供的宏列于表 B.5.18 中，它们表示可能的信号，可用作 raise()和 signal()的参数。当然，实现也可以添加更多的值。

表 B.5.18　信　号　宏

宏	描述
SIGABRT	异常终止，例如 abort()调用发出的信号
SIGFPE	错误的算术运算
SIGILL	检测到无效功能（例如，非法指令）
SIGINT	接收到交互注意信号（如，DOS 中断）
SIGSEGV	非法访问内存
SIGTERM	向程序发送终止请求

signal()函数的第 2 个参数接受一个指向 void 函数的指针，该函数有一个 int 类型的参数，也返

回相同类型的指针。为响应一个信号而被调用的函数称为**信号处理器**（*signal handler*）。标准定义了 3 个满足下面原型的宏：

```
void (*funct)(int);
```

表 B.5.19 列出了这 3 种宏。

表 B.5.19　void (*f)(int)宏

宏	描述
SIG_DFL	当该宏与一个信号值一起作为 signal()的参数时，表示默认处理信号
SIG_ERR	如果 signal()不能返回它的第 2 个参数，就用该宏作为它的返回值
SIG_IGN	当该宏与一个信号值一起作为 signal()的参数时，表示忽略信号

如果产生了信号 sig，而且 func 指向一个函数（参见表 B.5.20 中 signal()原型），那么大多数情况下先调用 signal(sig, SIG_DFL)把信号重置为默认设置，然后调用(*func)(sig)。可以执行返回语句或调用 abort()、exit()或 longjmp()来结束 func 指向的信号处理函数。

表 B.5.20　信 号 函 数

宏	描述
void (*signal(int sig, void (*func)(int)))(int);	如果产生信号 sig，则执行 func 指向的函数；如果能执行则返回 func，否则返回 SIG_ERR
int raise(int sig);	向执行程序发送信号 sig；如果成功发送则返回 0，否则返回非 0

B.5.13　对齐：stdalign.h（C11）

stdalign.h 头文件定义了 4 个宏，用于确定和指定数据对象的对齐属性。表 B.5.21 中列出了这些宏，其中前两个创建的别名与 C++的用法兼容。

表 B.5.21　void (*f)(int)宏

宏	描述
alignas	展开为关键字_Alignas
alignof	展开为关键字_Alignof
__alignas_is_defined	展开为整型常量 1，适用于#if
__alignof_is_defined	展开为整型常量 1，适用于#if

B.5.14　可变参数：stdarg.h

stdarg.h 头文件提供一种方法定义参数数量可变的函数。这种函数的原型有一个形参列表，列表中至少有一个形参后面跟有省略号：

```
void f1(int n, ...);                /* 有效 */
int f2(int n, float x, int k, ...); /* 有效 */
double f3(...);                     /* 无效 */
```

在下面的表中，*parmN* 是省略号前面的最后一个形参的标识符。在上面的例子中，第 1 种情况的 *parmN* 为 n，第 2 种情况的 *parmN* 为 k。

头文件中声明了 va_lis 类型表示储存形参列表中省略号部分的形参数据对象。表 B.5.22 中列出了 3

个带可变参数列表的函数中用到的宏。在使用这些宏之前要声明一个 va_list 类型的对象。

表 B.5.22 可变参数列表宏

宏	描述
void va_start(va_list ap, *parmN*);	该宏在 va_arg()和 va_end()使用 ap 之前初始化 ap，*parN* 是形参列表中最后一个形参名的标识符
void va_copy(va_list dest, va_list src);	该宏把 dest 初始化为 src 当前状态的备份（C99）
type va_arg(va_list ap, *type*);	该宏展开为一个表达式，其值和类型都与 ap 表示的形参列表的下一项相同，type 是该项的类型。每次调用该宏都前进到 ap 中的下一项
void va_end(va_list ap);	该宏关闭以上过程，可能导致 ap 在再次调用 va_start()之前不可用
void va_copy(va_list dest, va_list src);	该宏把 dest 初始化为 srt 当前状态的备份（C99）

B.5.15 原子支持：stdatomic.h（C11）

stdatomic.h 和 threads.h 头文件支持并发编程。并发编程的内容超过了本书讨论的范围，简单地说，stdatomic.h 头文件提供了创建原子操作的宏。编程社区使用原子这个术语是为了强调不可分割的特性。一个操作（如，把一个结构赋给另一个结构）从编程层面上看是原子操作，但是从机器语言层面上看是由多个步骤组成。如果程序被分成多个线程，那么其中的线程可能读或修改另一个线程正在使用的数据。例如，可以想象给一个结构的多个成员赋值，不同线程给不同成员赋值。有了 stdatomic.h 头文件，就能创建这些可以看作是不可分割的操作，这样就能保证线程之间互不干扰。

B.5.16 布尔支持：stdbool.h（C99）

stdbool.h 头文件定义了 4 个宏，如表 B.5.23 所列。

表 B.5.23 stdbool.h 宏

宏	描述
bool	展开为_Bool
false	展开为整型常量 0
true	展开为整型常量 1
__bool_true_false_ are_defined	展开为整型常量 1

B.5.17 通用定义：stddef.h

该头文件定义了一些类型和宏，如表 B.5.24 和表 B.5.25 所列。

表 B.5.24 stddef.h 类型

类型	描述
ptrdiff_t	有符号整数类型，表示两个指针之差
size_t	无符号整数类型，表示 sizeof 运算符的结果
wchar_t	整数类型，表示支持的本地化所指定的最大扩展字符集

表 B.5.25 stddef.h 宏

类型	描述
NULL	实现定义的常量，表示空指针
offsetof(type,*member-designator*)	展开为 size_t 类型的值，表示 type 类型结构的指定成员在该结构中的偏移量，以字节为单位。如果成员是一个位字段，该宏的行为是未定义的

示例

```
#include <stddef.h>

struct car
{
    char brand[30];
    char model[30];
    double hp;
    double price;
};
int main(void)
{
    size_t into = offsetof(struct car, hp); /* hp 成员的偏移量 */
    ...
```

B.5.18 整数类型：stdint.h

stdint.h 头文件中使用 typedef 工具创建整数类型名，指定整数的属性。stdint.h 头文件包含在 inttypes.h 中，后者提供输入/输出函数调用的宏。参考资料 VI 的“扩展的整数类型”中介绍了这些类型的用法。

1. 精确宽度类型

stdint.h 头文件中用一组 typedef 标识精确宽度的类型。表 B.5.26 列出了它们的类型名和大小。然而，注意，并不是所有的系统都支持其中的所有类型。

表 B.5.26 确切宽度类型

typedef 名	属性
int8_t	8 位，有符号
int16_t	16 位，有符号
int32_t	32 位，有符号
int64_t	64 位，有符号
uint8_t	8 位，无符号
uint16_t	16 位，无符号
uint32_t	32 位，无符号
uint64_t	64 位，无符号

2. 最小宽度类型

最小宽度类型保证其类型的大小至少是某数量位。表 B.5.27 列出了最小宽度类型，系统中一定会有这些类型。

表 B.5.27　最小宽度类型

typedef 名	属性
int_least8_t	至少 8 位，有符号
int_least16_t	至少 16 位，有符号
int_least32_t	至少 32 位，有符号
int_least64_t	至少 64 位，有符号
uint_least8_t	至少 8 位，无符号
uint_least16_t	至少 16 位，无符号
uint_least32_t	至少 32 位，无符号
uint_least64_t	至少 64 位，无符号

3. 最快最小宽度类型

在特定系统中，使用某些整数类型比其他整数类型更快。为此，stdint.h 也定义了最快最小宽度类型，如表 B.5.28 所列，系统中一定会有这些类型。

表 B.5.28　最快最小宽度类型

typedef 名	属性
int_fast8_t	至少 8 位有符号
int_fast16_t	至少 16 位有符号
int_fast32_t	至少 32 位有符号
int_fast64_t	至少 64 位有符号
uint_fast8_t	至少 8 位无符号
uint_fast16_t	至少 16 位无符号
uint_fast32_t	至少 32 位无符号
uint_fast64_t	至少 64 位无符号

4. 最大宽度类型

stdint.h 头文件还定义了最大宽度类型。这种类型的变量可以储存系统中的任意整数值，还要考虑符号。表 B.5.29 列出了这些类型。

表 B.5.29　最大宽度类型

typedef 名	描述
intmax_t	最大宽度的有符号类型
uintmax_t	最大宽度的无符号类型

5. 可储存指针值的整数类型

stdint.h 头文件中还包括表 B.5.30 中所列的两种整数类型，它们可以精确地储存指针值。也就是说，如果把一个 void *类型的值赋给这种类型的变量，然后再把该类型的值赋回给指针，不会丢失任何信息。系统可能不支持这类型。

表 B.5.30　可储存指针值的整数类型

typedef 名	描述
intptr_t	可储存指针值的有符号类型
uintptr_t	可储存指针值的无符号类型

6. 已定义的常量

stdint.h 头文件定义了一些常量，用于表示该头文件中所定义类型的限定值。常量都根据类型命名，即用_MIN 或_MAX 代替类型名中的_t，然后把所有字母大写即得到表示该类型最小值或最大值的常量名。例如，int32_t 类型的最小值是 INT32_MIN、unit_fast16_t 的最大值是 UNIT_FAST16_MAX。表 B.5.31 总结了这些常量以及与之相关的 intptr_t、unitptr_t、intmax_t 和 uintmax_t 类型，其中的 N 表示位数。这些常量的值应等于或大于（除非指明了一定要等于）所列的值。

表 B.5.31　整 型 常 量

常量标识符	最小值
INT*N*_MIN	等于-(2^{N-1}-1)
INT*N*_MAX	等于 2^{N-1}-1
UINT*N*_MAX	等于 2^{N-1}-1
INT_LEAST*N*_MIN	-(2^{N-1}-1)
INT_LEAST*N*_MAX	2^{N-1}-1
UINT_LEAST*N*_MAX	2^{N}-1
INT_FAST*N*_MIN	-(2^{N-1}-1)
INT_FAST*N*_MAX	2^{N-1}-1
UINT_FAS*N*_MAX	2^{N}-1
INTPTR_MIN	-(2^{15}-1)
INTPTR_MAX	2^{15}-1
UINTPTR_MAX	2^{16}-1
INTMAX_MIN	-(2^{15}-1)
INTMAX_MAX	2^{63}-1
UINTMAX_MAX	2^{64}-1

该头文件还定义了一些别处定义的类型使用的常量，如表 B.5.32 所示。

表 B.5.32　其他整型常量

常量标识符	含义
PTRDIFF_MIN	ptrdiff_t 类型的最小值
PTRDIFF_MAX	ptrdiff_t 类型的最大值
SIG_ATOMIC_MIN	sig_atomic_t 类型的最小值
SIG_ATOMIC_MAX	sig_atomic_t 类型的最大值
WCHAR_MIN	wchar_t 类型的最小值
WCHAR_MAX	wchar_t 类型的最大值
WINT_MIN	wint_t 类型的最小值
WINT_MAX	wint_t 类型的最大值
SIZE_MAX	size_t 类型的最大值

7. 扩展的整型常量

stdin.h 头文件定义了一些宏用于指定各种扩展整数类型。从本质上看，这种宏是底层类型（即在特定实现中表示扩展类型的基本类型）的强制转换。

把类型名后面的_t 替换成_C，然后大写所有的字母就构成了一个宏名。例如，使用表达式 UNIT_LEAST64_C(1000)后，1000 就是 unit_least64_t 类型的常量。

B.5.19 标准 I/O 库：stdio.h

ANSI C 标准库包含一些与流相关联的标准 I/O 函数和 stdio.h 头文件。表 B.5.33 列出了 ANSI 中这些函数的原型和简介（第 13 章详细介绍过其中的一些函数）。stdio.h 头文件定义了 FILE 类型、EOF 和 NULL 的值、标准 I/O 流（stdin、stdout 和 stderr）以及标准 I/O 库函数要用到的一些常量。

表 B.5.33 C 标准 I/O 函数

原型	描述
void clearerr(FILE *);	清除文件结尾和错误指示符
int fclose(FILE *);	关闭指定的文件
int feof(FILE *);	测试文件结尾
int ferror(FILE *);	测试错误指示符
int fflush(FILE *);	刷新指定的文件
int fgetc(FILE *);	获得指定输入流的下一个字符
int fgetpos(FILE *restrict, restrict);	储存文件位置指示符的 fpos_t *当前值
char * fgets(char *restrict, restrict);	从指定流中获取下一行（或 int、FILE *指定的字符数）
FILE * fopen(const char*restrict, const char*restrict);	打开指定的文件
int fprintf(FILE *restrict, const char *restrict, ...);	把格式化输出写入指定流
int fputc(int, FILE *);	把指定字符写入指定流
int fputs(const char* restrict, FILE * restrict);	把第 1 个参数指向的字符串写入指定流
size_t fread(void *restrict, size_t, size_t, FILE * restrict);	读取指定流中的二进制数据
FILE * freopen(const char * restrict, const char * restrict, FILE *restrict);	打开指定文件，并将其与指定流相关联
int fscanf(FILE *restrict, const char * restrict, ...);	读取指定流中的格式化输入
int fsetpos(FILE *,const fpos_t *);	设置文件位置指针指向指定的值
int fseek(FILE *, long,int);	设置文件位置指针指向指定的值
long ftell(FILE *);	获取当前文件位置
size_t fwrite(const void* restrict, size_t,size_t, FILE * restrict);	把二进制数据写入指定流
int getc(FILE *);	读取指定输入的下一个字符
int getchar();	读取标准输入的下一个字符
*char * gets(char *);*	获取标准输入的下一行（C11 库已删除）
void perror(const char*);	把系统错误信息写入标准错误中
int printf(const char *restrict, ...);	把格式化输出写入标准输出中

续表

原型	描述
int putc(int, FILE *);	把指定字符写入指定输出中
int putchar(int);	把指定字符写入指定输出中
int puts(const char *);	把字符串写入标准输出中
int remove(const char *);	移除已命名文件
int rename(const char *,constchar *);	重命名文件
void rewind(FILE *);	设置文件位置指针指向文件开始处
int scanf(const char *restrict, ...);	读取标准输入中的格式化输入
void setbuf(FILE *restrict, char * restrict);	设置缓冲区大小和位置
int setvbuf(FILE *restrict, char *restrict,int, size_t);	设置缓冲区大小、位置和模式
int snprintf(char *restrict, size_t n, const char * restrict, ...);	把格式化输出中的前 n 个字符写入指定字符串中
int sprintf(char *restrict, const char *restrict, ...);	把格式化输出写入指定字符串中
int sscanf(const char*restrict, const char *restrict, ...);	把格式化输入写入指定字符串中
FILE * tmpfile(void);	创建一个临时文件
char * tmpnam(char *);	为临时文件生成一个唯一的文件名
int ungetc(int, FILE *);	把指定字符放回输入流中
int vfprintf(FILE *restrict, const char *restrict, va_list);	与 fprintf()类似，但该函数用一个 va_list 类型形参列表（由 va_start 初始化）代替变量参数列表
int vprintf(const char *restrict, va_list);	与 printf()类似，但该函数用一个 va_list 类型形参列表（由 va_start 初始化）代替变量参数列表
int vsnprintf(char *restrict, size_t n); const char * restrict,va_list);	与 snprintf()类似，但该函数用一个 va_list 类型形参列表（由 va_start 初始化）代替变量参数列表
int vsprintf(char *restrict, const char *restrict, va_list);	与 sprintf()类似，但该函数用一个 va_list 类型形参列表（由 va_start 初始化）代替变量参数列表
int vscanf(const char *restrict, va list);	与 scanf()类似，但该函数用一个 va_list 类型形参列表（由 va_start 初始化）代替变量参数列表
int vsscanf(const char* restrict,* restrict,va_list);	与 sscanf()类似，但该函数用一个 va_list 类型形参列表（由 va_start 初始化）代替变量参数列表

B.5.20　通用工具：stdlib.h

ANSI C 标准库在 stdlib.h 头文件中定义了一些实用函数。该头文件定义了一些类型，如表 B.5.34 所示。

表 B.5.34　stdlib.h 中声明的类型

类型	描述
size_t	sizeof 运算符返回的整数类型
wchar_t	用于表示宽字符的整数类型
div_t	div()返回的结构类型，该类型中的 quot 和 rem 成员都是 int 类型
ldiv_t	ldiv()返回的结构类型，该类型中的 quot 和 rem 成员都是 long 类型
lldiv_t	lldiv()返回的结构类型，该类型中的 quot 和 rem 成员都是 long long 类型（C99）

stdlib.h 头文件定义的常量列于表 B.5.35 中。

表 B.5.35　stdlib.h 中定义的常量

类型	描述
NULL	空指针（相当于 0）
EXIT_FAILURE	可用作 exit()的参数，表示执行程序失败
EXIT_SUCCESS	可用作 exit()的参数，表示成功执行程序
RAND_MAX	rand()返回的最大值（一个整数）
MB_CUR_MAX	当前本地化的扩展字符集中多字节字符的最大字节数

表 B.5.36 列出了 stdlib.h 中的函数原型。

表 B.5.36　通 用 工 具

原型	描述
double atof(const char * nptr);	返回把字符串 nptr 开始部分的数字(和符号)字符转换为 double 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0
int atoi(const char* nptr);	返回把字符串 nptr 开始部分的数字（和符号）字符转换为 int 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0
int atol(const char* nptr);	返回把字符串 nptr 开始部分的数字（和符号）字符转换为 long 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0
double strtod(const char* restrict npt,char ** restrictept);	返回把字符串 npt 开始部分的数字(和符号)字符转换为 double 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0；如果转换成功，则把数字后第 1 个字符的地址赋给 ept 指向的位置；如果转换失败，则把 npt 赋给 ept 指向的位置
float strtof(const char * restrictnpt,char ** restrict ept);	与 strtod()类似，但是该函数把 npt 指向的字符串转换为 float 类型的值（C99）
long double strtols(const char * restrictnpt, char **restrict ept);	与 strtod()类似,但是该函数把 npt 指向的字符串转换成 long double 类型的值（C99）
long strtol(const char * restrict npt char ** restrict ept, int base);	返回把字符串 npt 开始部分的数字（和符号）字符转换成 long 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0；如果转换成功，则把数字后第 1 个字符的地址赋给 ept 指向的位置；如果转换失败，则把 npt 赋给 ept 指向的位置；假定字符串中的数字以 base 指定的数为基数
long long strtoll(const char *restrict npt, char** restrict ept,int base);	与 strtol()类似,但是该函数把 npt 指向的字符串转换为 long long 类型的值（C99）
unsigned long strtoul(const char * restrict npt, char** restrict ept, int base);	返回把字符串 npt 开始部分的数字（和符号）字符转换为 unsigned long 类型的值，跳过开始的空白，遇到第 1 个非数字字符时结束转换；如果未发现数字则返回 0；如果转换成功，则把数字后第 1 个字符的地址赋给 ept 指向的位置；如果转换失败，则把 npt 赋给 ept 指向的位置；假定字符串中的数字以 base 指定的数为基数

续表

原型	描述
unsigned long long strtoull(const char* restrict npt,char ** restrict ept, int base);	与 strtoul()类似，但是该函数把 npt 指向的字符串转换为 unsigned long long 类型的值（C99）
int rand(void);	返回 0～RAND_MAX 范围内的一个伪随机整数
void srand(unsigned int seed);	把随机数生成器种子设置为 seed，如果在调用 rand()之前调用 srand()，则种子为 1
void *aligned_alloc(size_t algn, size_t size);	为对齐对象 algn 分配 size 字节的空间，应支持 algn 对齐值，size 应该是 algn 的倍数（C11）
void *calloc(size_t nmem, size_t size);	为内含 nmem 个成员的数组分配空间，每个元素占 size 字节大；空间中的所有位都初始化为 0；如果操作成功，该函数返回数组的地址，否则返回 NULL
void free(void*ptr);	释放 ptr 指向的空间，ptr 应该是之前调用 calloc()、malloc()或 realloc()返回的值，或者 ptr 也可以是空指针，出现这种情况时什么也不做。如果 ptr 是其他值，其行为是未定义的
void *malloc(size_t size);	分配 size 字节的未初始化内存块；如果成功分配，该函数返回数组的地址，否则返回 NULL
void *realloc(void*ptr, size_t size);	把 ptr 指向的内存块大小改为 size 字节，size 字节内的内存块内容不变。该函数返回块的位置，它可能被移动。如果不能重新分配空间，函数返回 NULL，原始块不变；如果 ptr 为 NULL，其行为与调用带 size 参数的 malloc()相同；如果 size 是 0，且 ptr 不是 NULL，其行为与调用带 ptr 参数的 free()相同
void abort(void);	除非捕获信号 SIGABRT，且相应的信号处理器没有返回，否则该函数将导致程序异常结束。是否关闭 I/O 流和临时文件，因实现而异。该函数执行 raise(SIGABRT)
int atexit(void(*func)(void));	注册 func 指向的函数，使其在程序正常结束时被调用。实现应支持注册至少 32 个函数，并根据它们注册顺序的逆序调用。如果注册成功，函数返回 0；否则返回非 0
int at_quick_exit(void (*func)(void));	注册 func 指向的函数，如果调用 quick_exit()则调用被注册的函数。实现应支持注册至少 32 个函数，并根据它们注册顺序的逆序调用。如果注册成功，函数返回 0；否则返回非 0（C11）
void exit(int status);	该函数将正常结束程序。首先调用由 atexit()注册的函数，然后刷新所有打开的输出流、关闭所有的 I/O 流、关闭 tmpfile()创建的所有文件，并把控制权返回主机环境中；如果 status 是 0 或 EXIT_SUCCESS，则返回一个实现定义的值，表明未成功结束程序
void _Exit(int status);	与 exit()类似，但是该函数不调用 atexit()注册的函数和 signal()注册的信号处理器，其处理打开流的方式依实现而异
char *getenv(const char * name);	返回一个指向字符串的指针，该字符串表示 name 指向的环境变量的值。如果无法匹配指定的 name，则返回 NULL
_Noreturn void quick_exit(int status);	该函数将正常结束程序。不调用 atexit()注册的函数和 signal()注册的信号处理器。根据 at_quick_exit()注册函数的顺序，逆序调用这些函数。如果程序多次调用 quick_exit()或者同时调用 quick_exit()和 exit()，其行为是未定义的。通过调用_Exit(status)将控制权返回主机环境（C11）

续表

原型	描述
int system(const char *str);	把 str 指向的字符串传递给命令处理器（如 DOS 或 UNIX）执行的主机环境。如果 str 是 NULL 指针，且命令处理器可用，则该函数返回非 0，否则返回；如果 str 不是 NULL，返回值依实现而异
void *bsearch(const void *key, const void *base, size_tnmem, size_t size, int (*comp)(const void *, const void *));	查找 base 指向的一个数组（有 nmem 个元素，每个元素的大小为 size）中是否有元素匹配 key 指向的对象。通过 comp 指向的函数比较各项，如果 key 指向的对象小于数组元素，那么比较函数将返回小于 0 的值；如果两者相等，则返回 0；如果 key 指向的对象大于数组元素，则返回大于 0 的值。该函数返回指向匹配元素的指针或 NULL（如果无匹配元素）。如果有多个元素匹配，未定义返回哪一个元素
void qsort(void*base, size_t nmem, size_t size, int(*comp) (const void *, const void *));	根据 comp 指向的函数所提供的顺排列 base 指向的数组。该数组有 nmem 个元素，每个元素的大小是 size。如果第 1 个参数指向的对象小于数组元素，那么比较函数将返回小于 0 的值；如果两者相等，则返回 0；如果第 1 个参数指向的对象大于数组元素，则返回大于 0 的值
int abs(int n);	返回 n 的绝对值。如果 n 是负数但没有与之对应的正数，那么返回值是未定义的（当 n 是以二进制补码表示的 INT_MIN 时，会出现这种情况）
div_t div(int numer, int denom);	计算 number 除以 denom 的商和余，把商和余数分别储存在 div_t 结构的 quot 成员和 rem 成员中。对于无法整除的除法，商要趋零截断（即直接截去小数部分）
long labs(int n);	返回 n 的绝对值，如果 n 是负数但没有与之对应的正数，那么返回值是未定义的（当 n 是以二进制补码表示的 LONG_MIN 时，会出现这种情况）
ldiv_t ldiv(long numer, long denom);	计算 number 除以 denom 的商和余，把商和余数分别储存在 ldiv_t 结构的 quot 成员和 rem 成员中。对于无法整除的除法，商要趋零截断（即直接截去小数部分）
long long llabs(int n);	返回 n 的绝对值，如果 n 是负数但没有与之对应的正数，那么返回值是未定义的（当 n 是以二进制补码表示的 LONG_LONG_MIN 时，会出现这种情况）
lldiv_t lldiv(long numer, long denom);	计算 number 除以 denom 的商和余，把商和余数分别储存在 lldiv_t 结构的 quot 成员和 rem 成员中。对于无法整除的除法，商要趋零截断（即直接截去小数部分）（C99）
int mblen(const char *s, size_t n);	返回组成 s 指向的多字节字符的字节数（最大为 n）。如果 s 指向空字符，该函数则返回 0；如果 s 未指向多字节字符，则返回-1；如果 s 是 NULL，且多字节根据状态进行编码，该函数则返回非 0，否则返回 0
int mbtowc(wchar_t*pw, const char *s, size_t n);	如果 s 不是 NULL，该函数确定了组成 s 指向的多字节字符的字节数（最大为 n），并确定字符的 wchar_t 类型编码。如果 pw 不是 NULL，则把类型编码赋给 pw 指向的位置。返回值与 mblen(s, n) 相同
int wctomb(char *s,wchar_t wc);	把 wc 中的字符代码转换成相应的多字节字符表示，并将其储存在 s 指向的数组中，除非 s 是 NULL。如果 s 不是 NULL，且如果 wc 无法转换成相应的有效多字节字符，该函数返回-1；如果 wc 有效，该函数返回组成多字节的字节数；如果 s 是 NULL，且如果多字节字符根据状态进行编码，该函数则返回非 0，否则返回 0

续表

原型	描述
size_t mbstowcs(wchar_t *restrict pwcs,const char *srestrict ,size_t n);	把s指向的多字节字符数组转换成储存在pwcs开始位置的宽字符编码数组中，转换 pwcs 数组中的 n 个字符或转换到 s 数组的空字节停止。如果遇到无效的多字节字符，该函数返回(size_t)(-1);，否则返回已填充的数组元素个数（如果有空字符，不包含空字符）
size_t wcstombs(char * restricts, const wchart_t* restrict pwcs,size_t n);	把储存在 pwcs 指向数组中的宽字符编码序列转换成一个多字节字符序列，并把它拷贝到 s 指向的位置上，储存 n 个字节或遇到空字符时停止转换。如果遇到无效的宽字符编码，该函数返回(size_t)(-1)，否则返回已填充数组的字节数（如果有空字符，不包含空字符）

B.5.21　_Noreturn：stdnoreturn.h

stdnoreturn.h 定义了 noreturn 宏，该宏展开为_Noreturn。

B.5.22　处理字符串：string.h

string.h 库定义了 size_t 类型和空指针要使用的 NULL 宏。string.h 头文件提供了一些分析和操控字符串的函数，其中一些函数以更通用的方式处理内存。表 B.5.37 列出了这些函数。

表 B.5.37　字符串函数

原型	描述
void *memchr(const void *s, int c, size_t n);	在 s 指向对象的前 n 个字符中查找是否有 c。如果找到，则返回首次出现 c 处的指针，如果未找到则返回 NULL
int memcmp(const void*s1, const void *s2,size_t n);	比较 s1 指向对象中的前 n 个字符和 s2 指向对象的前 n 个字符，每个值都解释为 unsigned char 类型。如果 n 个字符都匹配，则两个对象完全相同；否则，比较两个对象中首次不匹配的字符对。如果两个对象相同，函数返回 0；如果在数值上第 1 个对象小于第 2 个对象，函数返回小于 0 的值；如果在数值上第 1 个对象大于第 2 个对象，函数返回大于 0 的值
void *memcpy(void *restrict s1, const void * restrict s2,size_t n);	把 s2 所指向位置上的 n 字节拷贝到 s1 指向的位置上，函数返回 s1 的值。如果两个位置出现重叠，其行为是未定义的
void *memmove(void*s1, const void *s2,size_t n);	把 s2 所指向位置上的 n 字节拷贝到 s1 指向的位置上，其行为与拷贝类似，返回 s1 的值。但是，如果出现局部重叠情况，该函数会先把重叠的内容拷贝至临时位置
void *memset(void *s,int v, size_t n);	把 v 的值（转换为 unsigned char）拷贝至 s 指向的前 n 字节中，函数返回 s
char *strcat(char *restrict s1, const char * restrict s2);	把 s2 指向的字符串拷贝到 s1 指向字符串后面，s2 字符串的第 1 个字符覆盖 s1 字符串的空字符。该函数返回 s1
char *strncat(char *restrict s1, const char * restrict s2,size_t n);	把 s2 指向字符串的 n 个字符拷贝到 s1 指向的字符串后面（或拷贝到 s2 的空字符为止）。s2 字符串的第 1 个字符覆盖 s1 字符串的空字符。函数返回 s1
char *strcpy(char *restrict s1, const char * restrict s2);	把 s2 指向的字符串拷贝到 s1 指向的位置。函数返回 s1

续表

原型	描述
char *strncpy(char *restrict s1, const char * restrict s2,size_t n);	把 s2 指向字符串的 n 个字符拷贝到 s1 指向的位置（或拷贝到 s2 的空字符为止）。如果在拷贝 n 个字符之前遇到空字符，则在拷贝字符后面添加若干个空字符，使其长度为 n；如果拷贝 n 个字符没有遇到空字符，则不添加空字符。函数返回 s1
int strcmp(const char*s1, const char *s2);	比较 s1 和 s2 指向的两个字符串。如果完全匹配，则两字符串相同，否则比较首次出现不匹配的字符对。通过字符编码值比较字符。如果两个字符串相同，函数返回 0；如果第 1 个字符串小于第 2 个字符串，函数返回小于 0 的值；如果第 1 个字符串大于第 2 个字符串，函数返回大于 0 的值
int strcoll(const char *s1, const char *s2);	与 strcmp() 类似，但是该函数使用当前本地化的 LC_COLLATE 类别（由 setlocale()函数设置）所指定的排序方式进行比较
int strncmp(const char *s1, const char *s2, size_t n);	比较 s1 和 s2 指向数组中的前 n 个字符，或比较到第 1 个空字符位置。如果所有的字符对都匹配，则两个数组相同否则比较两个数组中首次不匹配的字符对。通过字符编码值比较字符。如果两个数组相同，函数返回 0；如果第 1 个数组小于第 2 个数组，函数返回小于 0 的值；如果第 1 个数组大于第 2 个数组，函数返回大于 0 的值
size_t strxfrm(char* restrict s1, const char * restrict s2,size_t n);	转换 s2 中的字符串，并把转换后的前 n 个字符（包括空字符）拷贝到 s1 指向的数组中。用 strcmp()比较转换后的两个字符串的结果和用 strcoll()比较两个未转换字符串的结果相同。函数返回转换后的字符串长度（不包括末尾的空字符）
char *strchr(const char *s, int c);	查找 s 指向的字符串中首次出现 c 的位置。空字符是字符串的一部分。函数返回一个指针，指向首次出现 c 的位置。如果没有找到指定的 c 则返回 NULL
size_t strcspn(const char *s1, const char*s2);	返回 s1 中未出现 s2 中任何字符的最大起始段长度
char *strpbrk(const char *s1, const char*s2);	返回一个指针，指向 s1 中与 s2 任意字符匹配的第 1 个字符的位置。如果未发现任何匹配的字符，函数返回 NULL
char *strrchr(const char *s, int c);	在 s 指向的字符串中查找末次出现 c 的位置（即从 s2 右侧开始查找字符 c 首次出现的位置）。空字符是字符串的一部分。如果找到，函数返回指向该位置的指针；如果未找到，则返回 NULL
size_t strspn(const char *s1, const char*s2);	返回 s1 中包含 s2 所有字符的最大起始段长度
char *strstr(const char *s1, const char*s2);	返回一个指针，指向 s1 中首次出现 s2 中字符序列（不包括结束的空字符）的位置。如果未找到，函数返回 NULL
char *strtok(char *restrict s1, const char * restrict s2);	该函数把 s1 字符串分解为单独的记号。s2 字符串包含了作为记号分隔符的字符。按顺序调用该函数。第 1 次调用时，s1 应指向待分解的字符串。函数定位到非分隔符后的第 1 个记号分隔符，并用空字符替换它。函数返回一个指针，指向储存第 1 个记号的字符串。如果未找到记号，函数返回 NULL。在此次调用 strtok()查找字符串中的更多记号。每次调用都返回指向下一个记号的指针，如果未找到则返回 NULL（请参看表后面的示例）
char * strerror(int errnum);	返回一个指针，指向与储存在 errnum 中的错误号相对应的错误信息字符串（依实现而异）
int strlen(const char* s);	返回字符串 s 中的字符数（末尾的空字除外）

strtok()函数的用法有点不寻常，下面演示一个简短的示例。

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char data[] = " C is\t too#much\nfun!";
    const char tokseps[] = " \t\n#";/* 分隔符 */
    char * pt;

    puts(data);
    pt = strtok(data,tokseps);     /* 首次调用 */
    while (pt)                          /* 如果 pt 是 NULL，则退出 */
    {
        puts (pt);                      /* 显示记号 */
        pt = strtok(NULL, tokseps);/* 下一个记号 */
    }
    return 0;
}
```

下面是该示例的输出：

```
    C is too#much
fun!
C
is
too
much
fun!
```

B.5.23 通用类型数学：tgmath.h（C99）

math.h 和 complex.h 库中有许多类型不同但功能相似的函数。例如,下面 6 个都是计算正弦的函数：

```
double sin(double);
float sinf(float);
long double sinl(long double);
double complex csin(double complex);
float csinf(float complex);
long double csinl(long double complex);
```

tgmath.h 头文件定义了展开为通用调用的宏，即根据指定的参数类型调用合适的函数。下面的代码演示了使用 sin()宏时，展开为正弦函数的不同形式：

```
#include <tgmath.h>
...
double dx, dy;
float fx, fy;
long double complex clx, cly;
dy = sin(dx);               // 展开为 dy = sin(dx) （函数）
fy = sin(fx);               // 展开为 fy = sinf(fx)
cly = sin(clx);             // 展开为 cly = csinl(clyx)
```

tgmath.h 头文件为 3 类函数定义了通用宏。第 1 类由 math.h 和 complex.h 中定义的 6 个函数的变式组成，用 l 和 f 后缀和 c 前缀，如前面的 sin()函数所示。在这种情况下，通用宏名与该函数 double 类型版本的函数名相同。

第 2 类由 math.h 头文件中定义的 3 个函数变式组成，使用 l 和 f 后缀，没有对应的复数函数（如，erf()）。在这种情况下，宏名与没有后缀的函数名相同，如 erf()。使用带复数参数的这种宏的效果是未定义的。

第 3 类由 complex.h 头文件中定义的 3 个函数变式组成，使用 l 和 f 后缀，没有对应的实数函数，如 cimag()。使用带实数参数的这种宏的效果是未定义的。

表 B.5.38 列出了一些通用宏函数。

表 B.5.38　通用数学函数

acos	asin	atanb	acosh	asinh	atanh
cos	sin	tan	cosh	sinh	tanh
exp	log	pow	sqrt	fabs	atan2
cbrt	ceil	copysign	erf	erfc	exp2
expm1	fdim	floor	fma	fmax	fmin
fmod	frexp	hypot	ilogb	ldexp	lgamma
llrint	llround	log10	log1p	log2	logb
lrint	lround	nearbyint	nextafter	nexttoward	remainder
remquo	rint	round	scalbn	scalbln	tgamma
trunc	carg	cimag	conj	cproj	creal

在 C11 以前，编写实现必须依赖扩展标准才能实现通用宏。但是使用 C11 新增的_Generic 表达式可以直接实现。

B.5.24　线程：threads.h（C11）

threads.h 和 stdatomic.h 头文件支持并发编程。这方面的内容超出了本书讨论的范围，简而言之，该头文件支持程序执行多线程，原则上可以把多个线程分配给多个处理器处理。

B.5.25　日期和时间：time.h

time.h定义了 3 个宏。第 1 个宏是表示空指针的NULL，许多其他头文件中也定义了这个宏。第 2 个宏是CLOCKS_PER_SEC，该宏除以clock()的返回值得以秒为单位的时间值。第 3 个宏（C11）是TIME_UTC，这是一个正整型常量，用于指定协调世界时 [1]（即UTC）。该宏是timespec_get()函数的一个可选参数。

UTC 是目前主要世界时间标准，作为互联网和万维网的普通标准，广泛应用于航空、天气预报、同步计算机时钟等各领域。

time.h 头文件中定义的类型列在表 B.5.39 中。

表 B.5.39　time.h 中定义的类型

类型	描述
size_t	sizeof 运算符返回的整数类型
clock_t	适用于表示时间的算术类型
time_t	适用于表示时间的算术类型
struct timespec	以秒和纳秒为单位储存指定时间间隔的结构（C11）
struct tm	储存日历时间的各部分

timespec 结构中至少有两个成员，如表 B.5.40 所列。

[1] 也称为世界标准时间，简称 UTC，从英文“Coordinated Universal Time”/法文“Temps Universel Cordonné”而来。中国内地的时间与 UTC 的时差为+8，也就是 UTC+8。——译者注

表 B.5.40　timespec 结构中的成员

成员	描述
time_t tv_sec	秒（>=0）
long tv_nsec	纳秒（[0,999999999]）

日历类型的各组成部分被称为**分解时间**（*broken-down time*）。表 B.5.41 列出了 struct tm 结构中所需的成员。

表 B.5.41　struct tm 结构中的成员

成员	描述
int tm_sec	分后的秒（0-61）
int tm_min	小时后的分（0-59）
int tm_hour	小时（0-23）
int tm_mday	一个月的天数（0-31）
int tm_mon	一月后的月数（0-11）
int tm_year	1900 年后的年数
int tm_wday	星期日开始的天数（0-6）
int tm_yday	从 1 月 1 日开始的天数(0-365)
int tm_isdst	夏令时标志（大于 0 说明夏令时有效，等于 0 说明无效，小于 0 说明信息不可用）

日历时间（*calendar time*）表示当前的日期和时间，例如，可以是从 1900 年的第 1 秒开始经过的秒数。**本地时间**（*local time*）指的是本地时区的日历时间。表 B.5.42 列出了一些时间函数。

表 B.5.42　时 间 函 数

成员	描述
clock_t clock(void);	该函数返回实现从开始执行程序到调用该函数时，处理器经过的最接近的时间。该函数的返回值除以 CLOCK_PER_SEC 得到以秒为单位的时间。如果时间不可用或无法表示，函数返回(clock_t)(-1)
double difftime(time_t t1, time_t t0);	返回两个日历时间(t1 - t0)的差值。该函数返回计算结果，单位是秒
time_t mktime(struct tm *tmptr);	把 tmptr 指向的结构中的分解时间转换为日历时间。其编码与 time()函数相同，但是结构改变了，以便对结构中超出范围的值进行调整（例如，2 分 100 秒会调整为 4 分 40 秒），而且把 tm_wday 和 tm_yday 设置为其他成员指定的值。如果无法表示日历时间，该函数返回(time_t)(-1)；否则以 time_t 格式返回日历时间
time_t time(time_t *ptm)	返回当前日历时间，并将其储存在 ptm 指向的位置，假设 ptm 不是空指针。如果日期时间不可用，该函数返回(time_t)(-1)
int timespec_get(struct timespec * ts, int base)	根据指定的时基，把 ts 指向的结构设置为当前日历时间。如果成功，返回 base（非 0 值），否则返回 0（C11）
char *asctime(const struct tm *tmpt);	把 tmpt 指向的结构中的分解时间转换成 Thu Feb 26 13:14:33 1998\n\0 格式的字符串，并返回指向该字符串的指针

续表

成员	描述
char *ctime(const time_t*ptm);	把 ptm 指向的结构中的分解时间转换成 Wed Aug 11 10:48:24 1999\n\0 格式的字符串，并返回指向该字符串的指针
struct tm *gmtime(const time_t *ptm);	把 ptm 指向的日历时间转换成协调世界时（UTC）表示的分解时间，返回一个指向结构的指针，该结构中储时间信息。如果 UTC 不可用，则返回 NULL
struct tm*localtime(const time_t *ptm);	把 ptm 指向的日历时间转换成本地时间表示的分解时间，储存 tm 结构并返回指向该结构的指针
size_t strftime(char *restrict s, size_t max const char * restrict fmt, const struct tm *restrict tmpt);	把字符串 fmt 拷贝到字符串 s 中，用 tmp 指向的分解时间结构中的合适数据替换 fmt 中的转换说明（见表 B.5.43）。最多在 s 中放入 max 个字符。该函数返回放入 s 中的字符数（不包括空格）；如果字符串中的字符数大于 max，函数返回 0，且 s 中的内容不确定

表 B.5.43 列出了 strftime()函数中使用的转换说明。其中许多替换的值（如，月份名）都取决于当前的本地化设置。

表 B.5.43 strftime()函数中使用的转换说明

转换说明	被替换为
%a	本地化的星期名称缩写
%A	本地化的星期名称全名
%b	本地化的月份名称缩写
%B	本地化的月份名称全名
%c	本地化指定的日期和时间
%C	年份的后两位数字（年份除以 100，取小数部分的数）(00-99)
%d	十进制数表示的月份中的某天（01-31）
%D	月/日/年，等价于“%m/%d/%y”
%e	十进制数表示的月份中的某天，在仅一位的数字前有一个空格（ 1-31）
%F	年-月-日，等价于“%Y-%m-%d”
%g	基于周的年份的最后两位数字（00-99）
%G	十进制数表示的基于周的年份
%h	等价于“%b”
%H	十进制数（00-23）表示的小时（24 小时制）
%I	十进制数（01-12）表示的小时（12 小时制）
%j	十进制数表示的一年中的某天（001-366）
%m	十进制数表示的月份（01-12）
%n	换行符
%M	十进制数表示的分钟（00-59）
%p	等价于本地 12 小时制中的 am/pm
%r	本地的 12 小时制

续表

转换说明	被替换为
%R	小时:分钟，等价于“%H:%M”
%S	十进制数表示的秒（00-61）
%t	水平制表符
%T	小时:分钟：秒，等价于“%H:%M:%S”
%u	ISO 8601 的星期数（1～7），星期一为 1
%U	一年中的周数（00～53），把星期天作为一周的第 1 天
%V	ISO 8601 的一年周数（00～53），把星期天作为一周的第 1 天
%w	十进制表示的星期数（0～6），从星期天开始
%W	一年的周数（00～53），把星期一作为一周的第 1 天
%x	本地化日期表示
%X	本地化时间表示
%y	不带世纪的十进制年份（00～99）
%Y	带世纪的十进制年份
%z	按照 ISO 8601 格式的相对 UTC 偏移（“-800”表示格林威治时间后的 8 小时，即是向西 8 小时），如果无可用信息则无替换字符
%Z	时区名，如果无可用信息则无替换字符
%%	%（即百分号）

B.5.26　统一码工具：uchar.h（C11）

C99 的 wchar.h 头文件提供两种途径支持大型字符集。C11 专门针对统一码（Unicode）新增了适用于 UTF-16 和 UTF-32 编码的类型（见表 B.5.44）。

表 B.5.44　uchar.h 中声明的类型

类型	描述
char16_t	使用 16 位字符的无符号整数类型（与 stdint.h 中的 unit_least16_t 相同）
char32_t	使用 32 位字符的无符号整数类型（与 stdint.h 中的 unit_least32_t 相同）
size_t	sizeof 运算符(stddef.h)返回的整数类型
mbstate_t	非数组类型，可储存多字节字符序列和宽字符相互转换的转换状态信息

该头文件中还声明了一些多字节字符串与 char16_t、char32_t 格式相互转换的函数（见表 B.5.45）。

表 B.5.45　宽字符与多字节转换函数

类型	描述
size_t mbrto16(char16_t* restrict pwc, const char * restrict s, size_t n, mbstate_t* restrict ps);	与 mbrtowc()函数相同（wchar.h），但该函数是把字符转换为 char_16 类型，而不是 wchar_t 类型
size_t mbrto32(char32_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);	与 mbrto16()函数相同，但该函数是把字符转换为 char32_t 类型

续表

类型	描述
size_t c16rtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);	与 wcrtobm()函数相同（wchar.h），但该函数转换的是 char16_t 类型字符，而不是 wchar_t 类型
size_t c32rtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);	与 wcrtobm()函数相同（wchar.h），但该函数转换的是 char32_t 类型字符，而不是 wchar_t 类型

B.5.27 扩展的多字节字符和宽字符工具：wchar.h（C99）

每种实现都有一个基本字符集，要求 C 的 char 类型足够宽，以便能处理这个字符集。实现还要支持扩展的字符集，这些字符集中的字符可能需要多字节来表示。可以把多字节字符与单字节字符一起储存在普通的 char 类型数组，用特定的字节值指定多字节字符本身及其大小。如何解释多字节字符取决于**移位状态**（*shift state*）。在最初的移位状态中，单字节字符保留其通常的解释。特殊的多字节字符可以改变移位状态。除非显式改变特定的移位状态，否则移位状态一直保持有效。

wchar_t 类型提供另一种表示扩展字符的方法，该类型足够宽，可以表示扩展字符集中任何成员的编码。用这种宽字符类型来表示字符时，可以把单字符储存在 wchar_t 类型的变量中，把宽字符的字符串储存在 wchar_t 类型的数组中。字符的宽字符表示和多字节字符表示不必相同，因为后者可能使用前者并不使用的移位状态。

wchar.h 头文件提供了一些工具用于处理扩展字符的两种表示法。该头文件中定义的类型列在表 B.5.46 中（其中有些类型也定义在其他的头文件中）。

表 B.5.46 wchar.h 中定义的类型

类型	描述
wchar_t	整数类型，可表示本地化支持的最大扩展字符集
wint_t	整数类型，可储存扩展字符集的任意值和至少一个不是扩展字符集成员的值
size_t	sizeof 运算符返回的整数类型
mbstate_t	非数组类型，可储存多字节字符序列和宽字符之间转换所需的转换状态信息
struct tm	结构类型，用于储存日历时间的组成部分

wchar.h 头文件中还定义了一些宏，如表 B.5.47 所列。

表 B.5.47 wchar.h 中定义的宏

宏	描述
NULL	空指针
WCHAR_MAX	wchar_t 类型可储存的最大值
WCHAR_MIN	wchar_t 类型可储存的最小值
WEOF	wint_t 类型的常量表达式，不与扩展字符集的任何成员对；相当于 EOF 的宽字符表示，用于指定宽字符输入的文件结尾

该库提供的输入/输出函数类似于 stdio.h 中的标准输入/输出函数。在标准 I/O 函数返回 EOF 的情况中，对应的宽字符函数返回 WEOF。表 B.5.48 中列出了这些函数。

表 B.5.48　宽字符 I/O 函数

函数原型
int fwprintf(FILE * restrict stream, const wchar_t * restrict format, ...);
int fwscanf(FILE * restrict stream, const wchar_t * restrict format, ...);
int swprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, ...);
int swscanf(const wchar_t * restrict s, const wchar_t * restrict format,...);
int vfwprintf(FILE * restrict stream, const wchar_t * restrict format,va_list arg);
int vfwscanf(FILE * restrict stream, const wchar_t * restrict format,va_list arg);
int vswprintf(wchar_t * restrict s, size_t n, const wchar_t * restrict format, va_list arg);
int vswscanf(const wchar_t * restrict s, const wchar_t * restrict format,va_list arg);
int vwprintf(const wchar_t * restrict format, va_list arg);
int vwscanf(const wchar_t * restrict format, va_list arg);
int wprintf(const wchar_t * restrict format, ...);
int wscanf(const wchar_t * restrict format, ...);
wint_t fgetwc(FILE *stream);
wchar_t *fgetws(wchar_t * restrict s, int n, FILE * restrict stream);
wint_t fputwc(wchar_t c, FILE *stream);
int fputws(const wchar_t * restrict s, FILE * restrict stream);
int fwide(FILE *stream, int mode);
wint_t getwc(FILE *stream);
wint_t getwchar(void);
wint_t putwc(wchar_t c, FILE *stream);
wint_t putwchar(wchar_t c);
wint t ungetwc(wint_t c, FILE *stream);

有一个宽字符 I/O 函数没有对应的标准 I/O 函数：

int fwide(FILE *stream, int mode)[1];

如果 mode 为正，函数先尝试把形参表示的流指定为**宽字符定向**（*wide-charaacter oriented*）；如果 mode 为负，函数先尝试把流指定为**字节定向**（*byte oriented*）；如果 mode 为 0，函数则不改变流的定向。该函数只有在流最初无定向时才改变其定向。在以上所有的情况中，如果流是宽字符定向，函数返回正值；如果流是字节定向，函数返回负值；如果流没有定向，函数则返回 0。

wchar.h 头文件参照 string.h，也提供了一些转换和控制字符串的函数。一般而言，用 wcs 代替 sting.h 中的 str 标识符，这样 wcstod()就是 strtod()函数的宽字符版本。表 B.5.49 列出了这些函数。

表 B.5.49　宽字符字符串工具

函数原型
double wcstod(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
float wcstof(const wchar_t * restrict nptr, wchar_t ** restrict endptr);
long double wcstold(const wchar_t * restrict nptr, wchar_t ** restrict endptr);

[1] fwide()函数用于设置流的定向，根据 mode 的不同值来执行不同的工作。——译者注

续表

函数原型
long int wcstol(const wchar_t * restrict nptr, wchar_t ** restrict endptr,int base);
long long int wcstoll(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long int wcstoul(const wchar_t * restrict nptr, wchar_t ** restrict endptr, int base);
unsigned long long int wcstoull(const wchar_t * restrict nptr, wchar_t **restrict endptr, int base);
wchar_t *wcscpy(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncpy(wchar_t * restrict s1, const wchar_t * restrict s2, size_tn);
wchar_t *wcscat(wchar_t * restrict s1, const wchar_t * restrict s2);
wchar_t *wcsncat(wchar_t * restrict s1, const wchar_t * restrict s2, size_tn);
int wcscmp(const wchar_t *s1, const wchar_t *s2);
int wcscoll(const wchar_t *s1, const wchar_t *s2);
int wcsncmp(const wchar_t *s1, const wchar_t *s2, size_t n);
size_t wcsxfrm(wchar_t * restrict s1, const wchar_t * restrict s2, size_tn);
wchar_t *wcschr(const wchar_t *s, wchar_t c);
size_t wcscspn(const wchar_t *s1, const wchar_t *s2);
size_t wcslen(const wchar_t *s);
wchar_t *wcspbrk(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsrchr(const wchar_t *s, wchar_t c);
size_t wcsspn(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcsstr(const wchar_t *s1, const wchar_t *s2);
wchar_t *wcstok(wchar_t * restrict s1, const wchar_t * restrict s2, wchar_t** restrict ptr);
wchar_t *wmemchr(const wchar_t *s, wchar_t c, size_t n);
int wmemcmp(wchar_t * restrict s1, const wchar_t * restrict s2, size_t n);
wchar_t *wmemcpy(wchar_t * restrict s1,const wchar_t * restrict s2, size_t n);
wchar_t *wmemmove(wchar_t *s1, const wchar_t *s2, size_t n);
wchar_t *wmemset(wchar_t *s, wchar_t c, size_t n);

该头文件还参照 time.h 头文件中的 strtime()函数，声明了一个时间函数：

```
size_t wcsftime(wchar_t * restrict s, size_t maxsize,const wchar_t * restrict format,
const struct tm * restrict timeptr);
```

除此之外，该头文件还声明了一些用于宽字符字符串和多字节字符相互转换的函数，如表 B.5.50 所列。

表 B.5.50 宽字节和多字节字符转换函数

函数原型	描述
wint_t btowc(int c);	如果在初始移位状态中 c（unsigned char）是有效的单字节字符，那么该函数返回宽字节表示；否则，返回 WEOF
int wctob(wint_t c);	如果 c 是一个扩展字符集的成员，它在初始移位状态中的多字节字符表示的是单字节，该函数就返回一个转换为 int 类型的 unsigned char 的单字节表示；否则，函数返回 EOF
int mbsinit(const mbstate_t *ps);	如果 ps 是空指针或指向一个指定为初始转换状态的数据对象，函数就返回非零值；否则，函数返回 0

续表

函数原型	描述
size_t mbrlen(const char * restrict s, size_t n, mbstate_t * restrict ps);	mbrlen()函数相当于调用 mbrtowc(NULL, s, n, ps != NULL ? ps : &internal)，其中 internal 是 mbrlen()函数的 mbstate_t 对象，除非 ps 指定的表达式只计算一次
size_t mbrtowc(wchar_t * restrict pwc, const char * restrict s, size_t n, mbstate_t * restrict ps);	如果 s 是空指针，调用该函数相当于把 pwc 设置为空指针、把 n 设置为 1。如果 s 不是空指针，该函数最多检查 n 字节以确定下一个完整的多字节字符所需的字节数（包括所有的移位序列）。如果该函数确定了下一个多字节字符的结束处且合法，它就确定了对应宽字符的值。然后，如果 pwc 不为空，则把值储存在 pwc 指向的对象中。如果对应的宽字符是空的宽字符，描述的最终状态就是初始转换状态。如果检测到空的宽字符，函数返回 0；如果检测到另一个有效宽字符，函数返回完整字符所需的字节数。如果 n 字节不足以表示一个有效的宽字符，但是能表示其中的一部分，函数返回-2。如果出现编码错误，函数返回-1，并把 EILSEQ 储存在 errno 中，且不储存任何值
size_t wcrtomb(char * restrict s, wchar_t wc, mbstate_t * restrict ps);	如果 s 是空指针，那么调用该函数相当于把 wc 设置为空的宽字符，并为第 1 个参数使用内部缓冲区。如果 s 不是空指针，wcrtomb()函数则确定表示 wc 指定宽字符对应的多字节字符表示所需的字节数（包括所有移位序列），并把多字节字符表示储存在一个数组中（s 指向该数组的第 1 个元素），最多储存 MB_CUR_MAX 字节。如果 wc 是空的宽字符，就在初始移位状态所需的移位序列后储存一个空字节。描述的结果状态就是初始转换状态。如果 wc 是有效的宽字符，该函数返回储存多字节字符所需的字节数（包括指定移位状态的字节）。如果 wc 无效，函数则把 EILSEQ 储存在 errno 中，并返回-1
size_t mbsrtowcs(wchar_t * restrict dst, const char ** restrict src, size_t len, mbstate_t * restrict ps);	mbstrtows()函数把 src 间接指向的数组中的多字节字符序列转换成对应的宽字符序列，从 ps 指向的对象所描述的转换状态开始，一直转换到结尾的空字符（包括该字符并储存）或转换了 len 个宽字符。如果 dst 不是空指针，则待转换的字符将储存在 dst 指向的数组中。出现这两种情况时停止转换：如果字节序列无法构成一个有效的多字节字符，或者（如果 dst 不是空指针）len 个宽字符已储存在 dst 指向的数组中。每转换一次都相当于调用一次 mbrtowc()函数。如果 dst 不是空指针，就把空指针（如果因到达结尾的空字符而停止转换）或最后一个待转换多字节字符的地址赋给 src 指向的指针对象。如果由于到达结尾的空字符而停止转换，且 dst 不是空指针，那么描述的结果状态就是初始转状态。如果执行成功，函数返回成功转换的多字节字符数（不包括空字符）；否则函数返回-1
size_t wcsrtombs(char * restrict dst,const wchar_t ** restrict src,size_t len,mbstate_t * restrict ps);	wcsrtombs()函数把 src 间接指向的数组中的宽字符序列转换成对应的多字节字符序列（从 ps 指向的对象描述的转换状态开始）。如果 dst 不是空指针，待转换的字符将被储存在 dst 指向的数组中。一直转换到结尾的空字符（包括该字符并储存）或换了 len 个多字节字符。出现这两种情况时停止转换：如果宽字符没有对应的有效多字节字符，或者（如果 dst 不是空指针）下一个多字节字超过了储存在 dst 指向的数组中的总字节数 len 的限制。每转换一次都相当于调用一次 wcrtomb()函数。如果 dst 不是空指针，就把空指针（如果因到达结尾的空字符而停止转换）或最后一个待转换多字节字符的地址赋给 src 指向的指针对象。如果由于到达结尾的空字符而停止转换，描述的结果状态就是初始转状态。如果执行成功，函数返回成功转换的多字节字符数（不包括空字符）；否则函数返回-1

B.5.28 宽字符分类和映射工具：wctype.h（C99）

wctype.h 库提供了一些与 ctype.h 中的字符函数类似的宽字符函数，以及其他函数。wctype.h 还定义了表 B.5.51 中列出的 3 种类型和宏。

表 B.5.51　wctpe.h 中定义的类型和宏

类型/宏	描述
wint_t	整数类型，用于储存扩展字符集中的任意值，还可以储存至少一个不是扩展字符成员的值
wctrans_t	标量类型，可以表示本地化指定的字符映射
wctype_t	标量类型，可以表示本地化指定的字符分类
WEOF	wint_t 类型的常量表达式，不对应扩展字符集中的任何成员，相当于宽字符中的 EOF，用于表示宽字符输入的文件结尾

在该库中，如果宽字符参数满足字符分类函数的条件时，函数返回真（非 0）。一般而言，因为单字节字符对应宽字符，所以如果 ctype.h 中对应的函数返回真，宽字符函数也返回真。表 B.5.52 列出了这些函数。

表 B.5.52　宽字节分类函数

函数原型	描述
int iswalnum(wint_t wc);	如果 wc 表示一个字母数字字符（字母或数字），函数返回真
int iswalpha(wint_t wc);	如果 wc 表示一个字母字符，函数返回真
int iswblank(wint_t wc);	如果 wc 表示一个空格，函数返回真
int iswcntrl(wint_t wc);	如果 wc 表示一个控制字符，函数返回真
int iswdigit(wint_t wc);	如果 wc 表示一个数字，函数返回真
int iswgraph(wint_t wc);	如果 iswprint(wc)为真，且 iswspace(wc)为假，函数返回真
int iswlower(wint_t wc);	如果 wc 表示一个小写字符，函数返回真
int iswprint(wint_t wc);	如果 wc 表示一个可打印字符，函数返回真
int iswpunct(wint_t wc);	如果 wc 表示一个标点字符，函数返回真
int iswspace(wint_t wc);	如果 wc 表示一个制表符、空格或换行符，函数返回真
int iswupper(wint_t wc);	如果 wc 表示一个大写字符，函数返回真
int iswxdigit(wint_t wc);	如果 wc 表示一个十六进制数字，函数返回真

该库还包含两个**可扩展**的分类函数，因为它们使用当前本地化的 LC_CTYPE 值进行分类。表 B.5.53 列出了这些函数。

表 B.5.53　可扩展的宽字符分类函数

原型	描述
int iswctype(wint_t wc,wctype_t desc);	如果 wc 具有 desc 描述的属性，函数返回真
wctype_t wctype(const char *property);	wctype()函数构建了一个 wctpe_t 类型的值，它描述了由字符串参数 property 指定的宽字符分类。如果根据当前本地化的 LC_CTYPE 类别，property 识别宽字符分类有效，wctype()函数则返回非零值（可作为 iswctype()函数的第 2 个参数）；否则，函数返回 0

wctype()函数的有效参数名即是宽字符分类函数名去掉 isw 前缀。例如，wctype("alpha")表示的是 iswalpha()函数判断的字符类别。因此，调用 iswctype(wc, wctype("alpha"))相当于调用 iswalpha(wc)，唯一的区别是前者使用 LC_CTYPE 类别进行分类。

该库还有 4 个与转换相关的函数。其中有两个函数分别与 ctype.h 库中 toupper()和 tolower()相对应。第 3 个函数是一个可扩展的版本，通过本地化的 LC_CTYPE 设置确定字符是大写还是小写。第 4 个函数为第 3 个函数提供合适的分类参数。表 B.5.54 列出了这些函数。

表 B.5.54　宽字符转换函数

原型	描述
wint_t towlower(wint_t wc);	如果 wc 是大写字符，返回其小写形式；否则返回 wc
wint_t towupper(wint_t wc);	如果 wc 是小写字符，返回其大写形式；否则返回 wc
wint_t towctrans(wint_t wc, wctrans_t desc);	如果 desc 等于 wctrans("lower")的返回值，函数返回 wc 的小写形式（由 LC_CTYPE 设置确定）；如果 dest 等于 wctrans("upper")的返回值，函数返回 wc 的大写形式（由 LC_CTYPE 设置确定）
wctrans_t wctrans(const char*property);	如果参数是"lower"或"upper",函数返回一个 wctrans_t 类型的值，可用作 towctrans()的参数并反映 LC_CTYPE 设置，否则函数返回 0

B.6　参考资料 VI：扩展的整数类型

第 3 章介绍过，C99 的 inttypes.h 头文件为不同的整数类型提供一套系统的别名。这些名称与标准名称相比，能更清楚地描述类型的性质。例如，int 类型可能是 16 位、32 位或 64 位，但是 int32_t 类型一定是 32 位。

更精确地说，inttypes.h 头文件定义的一些宏可用于 scanf()和 printf()函数中读写这些类型的整数。inttypes.h 头文件包含的 stdlib.h 头文件提供实际的类型定义。格式化宏可以与其他字符串拼接起来形成合适格式化的字符串。

该头文件中的类型都使用 typedef 定义。例如，32 位系统的 int 可能使用这样的定义：

```
typedef int int32_t;
```

用#define 指令定义转换说明。例如，使用之前定义的 int32_t 的系统可以这样定义：

```
#define PRId32 "d" // 输出说明符
#define SCNd32 "d" // 输入说明符
```

使用这些定义，可以声明扩展的整型变量、输入一个值和显示该值：

```
int32_t cd_sales; // 32 位整数类型
scanf("%" SCNd32, &cd_sales);
printf("CD sales = %10" PRId32 " units\n", cd_sales);
```

如果需要，可以把字符串拼接起得到最终的格式字符串。因此，上面的代码可以这样写：

```
int cd_sales; // 32 位整数类型
scanf("%d", &cd_sales);
printf("CD sales = %10d units\n", cd_sales);
```

如果把原始代码移植到 16 位 int 的系统中，该系统可能把 int32_t 定义为 long，把 PRId32 定义为"ld"。但是，仍可以使用相同的代码，只要知道系统使用的是 32 位整型即可。

该参考资料的其余部分列出了扩展类型、转换说明以及表示类型限制的宏。

B.6.1 精确宽度类型

typedef 标识了一组精确宽度的类型，通用形式是 int*N*_t（有符号类型）和 uint*N*_t（无符号类型），其中 *N* 表示位数（即类型的宽度）。但是要注意，不是所有的系统都支持所有的这些类型。例如，最小可用内存大小是 16 位的系统就不支持 int8_t 和 uint8_t 类型。格式宏可以使用 d 或 i 表示有符号类型，所以 PRIi8 和 SCNi8 都有效。对于无符号类型，可以使用 o、x 或 u 以获得%o、%x 或%X 转换说明来代替%u。例如，可以使用 PRIX32 以十六进制格式打印 uint32_t 类型的值。表 B.6.1 列出了精确宽度类型、格式说明符和最小值、最大值。

表 B.6.1　精确宽度类型

类型名	printf()说明符	scanf()说明符	最小值	最大值
int8_t	PRId8	SCNd8	INT8_MIN	INT8_MAX
int16_t	PRId16	SCNd16	INT16_MIN	INT16_MAX
int32_t	PRId32	SCNd32	INT32_MIN	INT32_MAX
int64_t	PRId64	SCNd64	INT64_MIN	INT64_MAX
uint8_t	PRIu8	SCNu8	0	UINT8_MAX
uint16_t	PRIu16	SCNu16	0	UINT16_MAX
uint32_t	PRIu32	SCNu32	0	UINT32_MAX
uint64_t	PRIu64	SCNu64	0	UINT64_MAX

B.6.2 最小宽度类型

最小宽度类型保证一种类型的大小至少是某位。这些类型一定存在。例如，不支持 8 位单元的系统可以把 int_least_8 定义为 16 位类型。表 B.6.2 列出了最小宽度类型、格式说明符和最小值、最大值。

表 B.6.2　最小宽度类型

类型名	printf()说明符	scanf()说明符	最小值	最大值
int_least8_t	PRILEASTd8	SCNLEASTd8	INT_LEAST8_MIN	INT_LEAST8_MAX
int_least16_t	PRILEASTd16	SCNLEASTd16	INT_LEAST16_MIN	INT_LEAST16_MAX
int_least32_t	PRILEASTd32	SCNLEASTd32	INT_LEAST32_MIN	INT_LEAST32_MAX
int_least64_t	PRILEASTd64	SCNLEASTd64	INT_LEAST64_MIN	INT_LEAST64_MAX
uint_least8_t	PRILEASTu8	SCNLEASTu8	0	UINT_LEAST8_MAX
uint_least16_t	PRILEASTu16	SCNLEASTu16	0	UINT_LEAST16_MAX
uint_least32_t	PRILEASTu32	SCNLEASTu32	0	UINT_LEAST32_MAX
uint_least64_t	PRILEASTu64	SCNLEASTu64	0	UINT_LEAST64_MAX

B.6.3 最快最小宽度类型

对于特定的系统，用特定的整型更快。例如，在某些实现中 int_least16_t 可能是 short，但是系统在进行算术运算时用 int 类型会更快些。因此，inttypes.h 还定义了表示为某位数的最快类型。这些类型一定存在。在某些情况下，可能并未明确指定哪种类型最快，此时系统会简单地选择其中的一种。表 B.6.3 列出了最快最小宽度类型、格式说明符和最小值、最大值。

表 B.6.3　最快最小宽度类型

类型名	printf()说明符	scanf()说明符	最小值	最大值
int_fast8_t	PRIFASTd8	SCNFASTd8	INT_FAST8_MIN	INT_FAST8_MAX
int_fast16_t	PRIFASTd16	SCNFASTd16	INT_FAST16_MIN	INT_FAST16_MAX
int_fast32_t	PRIFASTd32	SCNFASTd32	INT_FAST32_MIN	INT_FAST32_MAX
int_fast64_t	PRIFASTd64	SCNFASTd64	INT_FAST64_MIN	INT_FAST64_MAX
uint_fast8_t	PRIFASTu8	SCNFASTu8	0	UINT_FAST8_MAX
uint_fast16_t	PRIFASTu16	SCNFASTu16	0	UINT_FAST16_MAX
uint_fast32_t	PRIFASTu32	SCNFASTu32	0	UINT_FAST32_MAX
uint_fast64_t	PRIFASTu64	SCNFASTu64	0	UINT_FAST64_MAX

B.6.4　最大宽度类型

有些情况下要使用最大整数类型，表 B.6.4 列出了这些类型。实际上，由于系统可能会提供比所需类型更大宽度的类型，因此这些类型的宽度可能比 long long 或 unsigned long long 更大。

表 B.6.4　最大宽度类型

类型名	printf()说明符	scanf()说明符	最小值	最大值
intmax_t	PRIdMAX	SCNdMAX	INTMAX_MIN	INTMAX_MAX
uintmax_t	PRIuMAX	SCBuMAX	0	UINTMAX_MAX

B.6.5　可储存指针值的整型

inttypes.h 头文件（通过包含 stdint.h 即可包含该头文件）定义了两种整数类型，可精确地储存指针值，见表 B.6.5。

表 B.6.5　可储存指针值的整数类型

类型名	printf()说明符	scanf()说明符	最小值	最大值
intptr_t	PRIdPTR	SCNdPTR	INTPTR_MIN	INTPTR_MAX
uintptr_t	PRIuPTR	SCBuPTR	0	UINTPTR_MAX

B.6.6　扩展的整型常量

在整数后面加上 L 后缀可表示 long 类型的常量，如 445566L。如何表示 int32_t 类型的常量？要使用 inttypes.h 头文件中定义的宏。例如，表达式 INT32_C(445566)展开为一个 int32_t 类型的常量。从本质上看，这种宏相当于把当前类型强制转换成底层类型，即特殊实现中表示 int32_t 类型的基本类型。

宏名是把相应类型名中的_C 用_t 替换，再把名称中所有的字母大写。例如，要把 1000 设置为 unit_least64_t 类型的常量，可以使用表达式 UNIT_LEAST64_C(1000)。

B.7　参考资料 VII：扩展字符支持

C 语言最初并不是作为国际编程语言设计的，其字符的选择或多或少是基于标准的美国键盘。但是，随着后来 C 在世界范围内越来越流行，不得不扩展来支持不同且更大的字符集。这部分参考资料概括介绍了一些相关内容。

B.7.1　三字符序列

有些键盘没有 C 中使用的所有符号，因此 C 提供了一些由三个字符组成的序列（即**三字符序列**）作为这些符号的替换表示。如表 B.7.1 所示。

表 B.7.1　三字符序列

三字符序列	符号	三字符序列	符号	三字符序列	符号
??=	#	??(	[	??/	\
??)	]	??'	^	??<	{
??!	|	??>	}	??-	～

C 替换了源代码文件中的这些三字符序列，即使它们在双引号中也是如此。因此，下面的代码：

```
??=include <stdio.h>
??=define LIM 100
int main()
??<
    int q??(LIM??);
    printf("More to come.??/n");
    ...
??>
```

会变成这样：

```
#include <stdio.h>
#define LIM 100
int main()
{
    int q[LIM];
    printf("More to come.\n");
    ...
}
```

当然，要在编译器中设置相关选项才能激活这个特性。

B.7.2　双字符

意识到三字符系统很笨拙，C99 提供了**双字符**（*digraph*），可以使用它们来替换某些标准 C 标点符号。

表 B.7.2　双　字　符

双字符	符号	双字符	符号	双字符	符号
<:	[	:>	]	<%	{
%>	}	%:	#	%:%:	##

与三字符不同的是，不会替换双引号中的双字符。因此，下面的代码：

```
%:include <stdio.h>
%:define LIM 100
int main()
<%
    int q<:LIM:>;
    printf("More to come.:>");
    ...
%>
```

会变成这样：

```
#include <stdio.h>
#define LIM 100
int main()
{
    int q[LIM];
    printf("More to come.:>"); // :>是字符串的一部分
    ...
}                              // :>与 }相同
```

B.7.3　可选拼写：iso646.h

使用三字符序列可以把||运算符写成??!??!，这看上去比较混乱。C99 通过 iso646.h 头文件（参考资料 V 中的表 B.5.11）提供了可展开为运算符的宏。C 标准把这些宏称为可选拼写（*alternative spelling*）。

如果包含了 iso646.h 头文件，以下代码：

```
if(x == M1 or x == M2)
    x and_eq 0XFF;
```

可展开为下面的代码：

```
if(x == M1 || x == M2)
    x &= 0XFF;
```

B.7.4　多字节字符

C 标准把多字节字符描述为一个或多个字节的序列，表示源环境或执行环境中的扩展字符集成员。源环境指的是编写源代码的环境，执行环境指的是用户运行已编译程序的环境。这两个环境不同。例如，可以在一个环境中开发程序，在另一个环境中运行该程序。扩展字符集是 C 语言所需的基本字符集的超集。

有些实现会提供扩展字符集，方便用户通过键盘输入与基本字符集不对应的字符。这些字符可用于字符串字面量和字符常量中，也可出现在文件中。有些实现会提供与基本字符集等效的多字节字符，可替换三字符和双字符。

例如，德国的一个实现也许会允许用户在字符串中使用日耳曼元音变音字符：

```
puts("eins zwei drei vier fünf");
```

一般而言，程序可使用的扩展字符集因本地化设置而异。

B.7.5　通用字符名（UCN）

多字节字符可以用在字符串中，但是不能用在标识符中。C99 新增了**通用字符名**（UCN），允许用户在标识名中使用扩展字符集中的字符。系统扩展了转义序列的概念，允许编码 ISO/IEC 10646 标准中的字符。该标准由国际标准化组织（ISO）和国际电工技术委员会（IEC）共同制定，为大量的字符提供数值码。10646 标准和统一码（*Unicode*）关系密切。

有两种形式的 UCN 序列。第 1 种形式是\u *hexquard*，其中 *hexquard* 是一个 4 位的十六进制数序列（如，\u00F6）。第 2 种形式是\U *hexquardhexquard*，如\U0000AC01。因为十六进制每一位上的数对应 4 位，\u 形式可用于 16 位整数表示的编码，\U 形式可用于 32 位整数表示的编码。

如果系统实现了 UCN，而且包含了扩展字符集中所需的字符，就可以在字符串、字符常量和标识符中使用 UCN：

```
wchar_t value\u00F6\u00F8 = L'\u00f6';
```

统一码和 ISO 10646

统一码为表示不同的字符集提供了一种解决方案，可以根据类型为大量字符和符号制定标准的编号系统。例如，ASCII 码被合并为统一码的子集，因此美国拉丁字符（如 A ~ Z）在这两个系统中的编码相同。但是，统一码还合并了其他拉丁字符（如，欧洲语言中使用的一些字符）和其他语言中的字符，包括希腊文、西里尔字母、希伯来文、切罗基文、阿拉伯文、泰文、孟加拉文和形意文字（如中文和日文）。到目前为止，统一码表示的符号超过了 110000 个，而且仍在发展中。欲了解更多细节，请查阅统一码联合站点：www.unicode.org。

统一码为每个字符分配一个数字，这个数字称为代码点（*code point*）。典型的统一码代码点类似：U-222B。U 表示该字符是统一字符，222B 是表示该字符的一个十六进制数，在这种情况下，表示积分号。

国际标准化组织（ISO）组建了一个团队开发 ISO 10646 和标准编码的多语言文本。ISO 10646 团队和统一码团队从 1991 年开始合作，一直保持两个标准的相互协调。

B.7.6 宽字符

C99 为使用宽字符提供更多支持，通过 `wchar.h` 和 `wctype.h` 库包含了更多大型字符集。这两个头文件把 `wchar_t` 定义为一种整型类型，其确切的类型依赖实现。该类型用于储存扩展字符集中的字符，扩展字符集是是基本字符集的超集。根据定义，`char` 类型足够处理基本字符集，而 `wchar_t` 类型则需要更多位才能储存更大范围的编码值。例如，`char` 可能是 `8` 位字节，`wchar_t` 可能是 `16` 位的 `unsigned short`。

用 `L` 前缀标识宽字符常量和字符串字面量，用`%lc` 和`%ls` 显示宽字符数据：

```
wchar_t wch = L'I';
wchar_t w_arr[20] = L"am wide!";
printf("%lc %ls\n", wch, w_arr);
```

例如，如果把 `wchar_t` 实现为 2 字节单元，`'I'`的 1 字节编码应储存在 `wch` 的低位字节。不是标准字符集中的字符可能需要两个字节储存字符编码。例如，可以使用通用字符编码表示超出 `char` 类型范围的字符编码：

```
wchar_t w = L'\u00E2'; /* 16位编码值 */
```

内含 `wchar_t` 类型值的数组可用于储存宽字符串，每个元素储存一个宽字符编码。编码值为 `0` 的 `wchar_t` 值是空字符的 `wchar_t` 类型等价字符。该字符被称为**空宽字符**（*null wide character*），用于表示宽字符串的结尾。

可以使用`%lc` 和`%ls` 读取宽字符：

```
wchar_t wch1;
wchar_t w_arr[20];
puts("Enter your grade:");
scanf("%lc", &wch1);
puts("Enter your first name:");
scanf("%ls",w_arr);
```

`wchar_t` 头文件为宽字符提供更多支持，特别是提供了宽字符 I/O 函数、宽字符转换函数和宽字符串控制函数。例如，可以用 `fwprintf()`和 `wprintf()`函数输出，用 `fwscanf()`和 `wscanf()`函数输入。与一般输入/输出函数的主要区别是，这些函数需要宽字符格式字符串，处理的是宽字符输入/输出流。例如，下面的代码把信息作为宽字符显示：

```
wchar_t * pw = L"Points to a wide-character string";
int dozen = 12;
wprintf(L"Item %d: %ls\n", dozen, pw);
```

类似地，还有 getwchar()、putwchar()、fgetws()和 fputws()函数。wchar_t 头文件定义了一个 WEOF 宏，与 EOF 在面向字节的 I/O 中起的作用相同。该宏要求其值是一个与任何有效字符都不对应的值。因为 wchar_t 类型的值都有可能是有效字符，所以 wchar_t 库定义了一个 wint_t 类型，包含了所有 wchar_t 类型的值和 WEOF 的值。

该库中还有与 string.h 库等价的函数。例如，wcscpy(ws1, ws2)把 ws1 指定的宽字符串拷贝到 ws2 指向的宽字符数组中。类似地，wcscmp()函数比较宽字符串，等等。

wctype.h 头文件新增了字符分类函数，例如，如果 iswdigit()函数的宽字符参数是数字，则返回真；如果 iswblank()函数的参数是空白，则返回真。空白的标准值是空格和水平制表符，分别写作 L'' 和 L'\t'。

C11 标准通过 uchar.h 头文件为宽字符提供更多支持，为匹配两种常用的统一码格式，定义了两个新类型。第 1 种类型是 char16_t，可储存一个 16 位编码，是可用的最小无符号整数类型，用于 *hexquard* UCN 形式和统一码 UTF-16 编码方案。

```
char16_t = '\u00F6';
```

第 2 种类型是 char32_t，可储存一个 32 位编码，最小的可用无符号整数类型，。可用于 *hexquard* UCN 形式和统一码 UTF-32 编码方案

```
char32_t = '\U0000AC01';
```

前缀 u 和 U 分别表示 char16_t 和 char32_t 字符串。

```
char16_t ws16[11] = u"Tannh\u00E4user";
char32_t ws32[13] = U"caf\U000000E9 au lait";
```

注意，这两种类型比 wchar_t 类型更具体。例如，在一个系统中，wchar_t 可以储存 32 位编码，但是在另一个系统中也许只能储存 16 位的编码。另外，这两种新类型都与 C++兼容。

B.7.7　宽字符和多字节字符

宽字符和多字节字符是处理扩展字符集的两种不同的方法。例如，多字节字符可能是一个字节、两个字节、三个字节或更多字节，而所有的宽字符都只有一个宽度。多字节字符可能使用移位状态（移位状态是一个字节，确定如何解释后续字节）；而宽字符没有移位状态。可以把多字节字符的文件读入使用标准输入函数的普通 char 类型数组，把宽字节的文件读入使用宽字符输入函数的宽字节数组。

C99 在 wchar.h 库中提供了一些函数，用于多字节和宽字节之间的转换。mbrtowc()函数把多字节字符转换为宽字符，wcrtomb()函数把宽字符转换为多字节字符。类似地，mbstrtowcs()函数把多字节字符串转换为宽字节字符串，wcstrtombs()函数把宽字节字符串转换为多字节字符串。

C11 在 uchar.h 库中提供了一些函数，用于多字节和 char16_t 之间的转换，以及多字节和 char32_t 之间的转换。

B.8　参考资料 VIII：C99/C11 数值计算增强

过去，FORTRAN 是数值科学计算和工程计算的首选语言。C90 使 C 的计算方法更接近于 FORTRAN。例如，float.h 中使用的浮点特性规范都是基于 FORTRAN 标准委员会开发的模型。C99 和 C11 标准继续增强了 C 的计算能力。例如，C99 新增的变长数组（C11 成为可选的特性），比传统的 C 数组更符合 FORTRAN 的用法（如果实现不支持变长数组，C11 指定了__STDC_NO_VLA__宏的值为 1）。

B.8.1　IEC 浮点标准

国际电工技术委员会（IEC）已经发布了一套浮点计算的标准（IEC 60559）。该标准包括了浮点数的格式、精度、NaN、无穷值、舍入规则、转换、异常以及推荐的函数和算法等。C99 纳入了该标准，将其作为 C 实现浮点计算的指导标准。C99 新增的大部分浮点工具（如，`fenv.h` 头文件和一些新的数学函数）都基于此。另外，`float.h` 头文件定义了一些与 IEC 浮点模型相关的宏。

1. 浮点模型

下面简要介绍一下浮点模型。标准把浮点数 x 看作是一个基数的某次幂乘以一个分数，而不是 C 语言的 E 记数法（例如，可以把 876.54 写成 0.87654E3）。正式的浮点表示更为复杂：

$$x = sb^e \sum_{k=1}^{p} f_k^{\,b^{-k}}$$

简单地说，这种表示法把一个数表示为**有效数**（*significand*）与 b 的 e 次幂的乘积。

下面是各部分的含义。

s 代表符号（±1）。

b 代表基数。最常见的值是 2，因为浮点处理器通常使用二进制数学。

e 代表整数指数（不要与自然对数中使用的数值常量 e 混淆），限制最小值和最大值。这些值依赖于留出储存指数的位数。

f_k 代表基数为 b 时可能的数字。例如，基数为 2 时，可能的数字是 0 和 1；在十六进制中，可能的数字是 0～F。

p 代表精度，基数为 b 时，表示有效数的位数。其值受限于预留储存有效数字的位数。

明白这种表示法的关键是理解 `float.h` 和 `fenv.h` 的内容。下面，举两个例子解释内部如何表示浮点数。

首先，假设一个浮点数的基数 b 为 10，精度 p 为 5。那么，根据上面的表示法，24.51 应写成：

$(+1)10^3(2/10 + 4/100 + 5/1000 + 1/10000 + 0/100000)$

假设计算机可储存十进制数（0～9），那么可以储存符号、指数 3 和 5 个 f_k 值：2、4、5、1、0（这里，f_1 是 2，f_2 是 4，等等）。因此，有效数是 0.24510，乘以 10^3 得 24.51。

接下来，假设符号为正，基数 b 是 2，p 是 7（即，用 7 位二进制数表示），指数是 5，待储存的有效数是 1011001。下面，根据上面的公式构造该数：

$$\begin{aligned} x &= (+1)2^5\ (1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128) \\ &= 32(1/2 + 0/4 + 1/8 + 1/16 + 0/32 + 0/64 + 1/128) \\ &= 16 + 0 + 4 + 2 + 0 + 0 + 1/4 = 22.25 \end{aligned}$$

`float.h` 中的许多宏都与该浮点表示相关。例如，对于一个 `float` 类型的值，表示基数的 FLT_RADIX 是 b，表示有效数位数（基数为 b 时）的 FLT_MANT_DIG 是 p。

2. 正常值和低于正常的值

正常浮点值（*normalized floating-point value*）的概念非常重要，下面简要介绍一下。为简单起见，先假设系统使用十进制（b = FLT_RADIX = 10）和浮点值的精度为 5（p = FLT_MANT_DIG = 5）（标准要求的精度更高）。考虑下面表示 31.841 的方式：

指数 = 3，**有效数** = .31841（.31841E3）

指数 = 4，有效数 = .03184（.03184E4）
指数 = 5，有效数 = .00318（.00318E5）

显而易见，第 1 种方法精度最高，因为在有效数中使用了所有的 5 位可用位。规范化浮点非零值是第 1 位有效位为非零的值，这也是通常储存浮点数的方式。

现在，假设最小指数（FLT_MIN_EXP）是-10，那么最小的规范值是：

指数 = -10，有效数 = .10000（.10000E-10）

通常，乘以或除以 10 意味着使指数增大或减小，但是在这种情况下，如果除以 10，却无法再减小指数。但是，可以改变有效数获得这种表示：

指数 = -10，有效数 = .01000（.01000E-10）

这个数被称为**低于正常的**（*subnormal*），因为该数并未使用有效数的全精度。例如，0.12343E-10 除以 10 得.01234E-10，损失了一位的信息。

对于这个特例，0.1000E-10 是最小的非零正常值（FLT_MIN），最小的非零低于正常值是 0.00001E-10（FLT_TRUE_MIN）。

float.h 中的宏 FLT_HAS_SUBNURM、DBL_HAS_SUBNORM 和 LDBL_HAS_SUBNORM 表征实现如何处理低于正常的值。下面是这些宏可能会用到的值及其含义：

-1	不确定（尚未统一）
0	不存在（例如，实现可能会用 0 替换低于正常的值）
1	存在

math.h 库提供一些方法，包括 fpclassify()和 isnormal()宏，可以识别程序何时生成低于正常的值，这样会损失一些精度。

3. 求值方案

float.h 中的宏 FLT_EVAL_METHOD 确定了实现采用何种浮点表达式的求值方案，如下所示（有些实现还会提供其他负值选项）。

-1	不确定
0	对在所有浮点类型范围和精度内的操作、常量求值
1	对在 double 类型的精度内和 float、double 类型的范围内的操作、常量求值，对 longdouble 范围内的 long double 类型的操作、常量求值
2	对所有浮点类型范围内和 long double 类型精度内的操作和常量求值

例如，假设程序中要把两个 float 类型的值相乘，并把乘积赋给第 3 个 float 类型变量。对于选项 1（即 K&R C 采用的方案），这两个 float 类型的值将被扩展为 double 类型，使用 double 类型完成乘法计算，然后在赋值计算结果时再把乘积转为 float 类型。

如果选择 0（即 ANSI C 采用的方案），实现将直接使用这两个 float 类型的值相乘，然后赋值乘积。这样做比选项 1 快，但是会稍微损失一点精度。

4. 舍入

float.h 中的宏 FLT_ROUNDS 确定了系统如何处理舍入，其指定值所对应的舍入方案如下所示。

-1	不确定
0	趋零截断
1	舍入到最接近的值
2	趋向正无穷

3　　　　趋向负无穷

系统可以定义其他值，对应其他舍入方案。

一些系统提供控制舍入的方案，在这种情况下，fenv.h 中的 festround()函数提供编程控制。

如果只是计算制作 37 个蛋糕需要多少面粉，这些不同的舍入方案可能并不重要，但是对于金融和科学计算而言，这很重要。显然，把较高精度的浮点值转换成较低精度值时需要使用舍入方案。例如，把 double 类型的计算结果赋给 float 类型的变量。另外，在改变进制时，也会用到舍入方案。不同进制下精确表示的分数不同。例如，考虑下面的代码：

```
float x = 0.8;
```

在十进制下，8/10 或 4/5 都可以精确表示 0.8。但是大部分计算机系统都以二进制储存结果，在二进制下，4/5 表示为一个无限循环小数：

```
0.1100110011001100...
```

因此，在把 0.8 储存在 x 中时，将其舍入为一个近似值，其具体值取决于使用的舍入方案。

尽管如此，有些实现可能不满足 IEC 60559 的要求。例如，底层硬件可能无法满足要求。因此，C99 定义了两个可用作预处理器指令的宏，检查实现是否符合规范。第 1 个宏是__STDC_IEC_559__，如果实现遵循 IEC 60559 浮点规范，该宏被定义为常量 1。第 2 个宏是__STDC_IEC_559_COMPLEX__，如果实现遵循 IEC 60559 兼容复数运算，该宏被定义为常量 1。

如果实现中未定义这两个宏，则不能保证遵循 IEC 60559。

B.8.2　fenv.h 头文件

fenv.h 头文件提供一些与浮点环境交互的方法。也就是说，允许用户设置浮点**控制模式值**（该值管理如何执行浮点运算）并确定浮点状态标志（或异常）的值（报告运算效果的信息）。例如，控制模式设置可指定舍入的方案；如果运算出现浮点溢出则设置一个状态标志。设置状态标志的操作叫作**抛出异常**。

状态标志和控制模式只有在硬件支持的前提下才能发挥作用。例如，如果硬件没有这些选项，则无法更改舍入方案。

使用下面的编译指示开启支持：

```
#pragma STDC FENV_ACCESS ON
```

这意味着程序到包含该编译指示的块末尾一直支持，或者如果该编译指示是外部的，则支持到该文件或翻译单元的末尾。使用下面的编译指示关闭支持：

```
#pragma STDC FENV_ACCESS OFF
```

使用下面的编译指示可恢复编译器的默认设置，具体设置取决于实现：

```
#pragma STDC FENV_ACCESS DEFAULT
```

如果涉及关键的浮点运算，这个功能非常重要。但是，一般用户使用的程度有限，所以本附录不再深入讨论。

B.8.3　STDC FP_CONTRACT 编译指示

一些浮点数处理器可以把有多个运算符的浮点表达式合并成一个运算。例如，处理器只需一步就求出下面表达式的值：

```
x*y - z
```

这加快了运算速度，但是减少了运算的可预测性。STDC FP_CONTRACT 编译指示允许用户开启或关闭这个特性。默认状态取决于实现。

为特定运算关闭合并特性，然后再开启，可以这样做：

```
#pragma STDC FP_CONTRACT OFF
val = x * y - z;
#pragma STDC FP_CONTRACT ON
```

B.8.4　math.h 库增补

大部分 C90 数学库中都声明了 double 类型参数和 double 类型返回值的函数，例如：

```
double sin(double);
double sqrt(double);
```

C99 和 C11 库为所有这些函数都提供了 float 类型和 long double 类型的函数。这些函数的名称由原来函数名加上 f 或 l 后缀构成，例如：

```
float sinf(float);                  /* sin()的 float 版本 */
long double sinl(long double);      /* sin()的 long double 版本 */
```

有了这些不同精度的函数系列，用户可以根据具体情况选择最效率的类型和函数组合。

C99 还新增了一些科学、工程和数学运算中常用的函数。表 B.5.16 列出了所有数学函数的 double 版本。在许多情况下，这些函数的返回值都可以使用现有的函数计算得出，但是新函数计算得更快更精确。例如，loglp(x)表示的值与与 log(1 + x)相同，但是 loglp(x)使用了不同的算法，对于较小的 x 值而言计算更精确。因此，可以使用 log()函数作普通运算，但是对于精确要求较高且 x 值较小时，用 loglp()函数更好。

除这些函数以外，数学库中还定义了一些常量和与数字分类、舍入相关的函数。例如，可以把值分为无穷值、非数（NaN）、正常值、低于正常的值、真零。[NaN 是一个特别的值，用于表示一个不是数的值。例如，asin(2.0)返回 NaN，因为定义了 asin()函数的参数必须是-1～1 范围内的值。低于正常的值是比使用全精度表示的最小值还要小的数。]还有一些专用的比较函数，如果一个或多个参数是非正常值时，函数的行为与标准的关系运算符不同。

使用 C99 的分类方案可以检测计算的规律性。例如，math.h 中的 isnormal()宏，如果其参数是一个正常的数，则返回真。下面的代码使用该宏在 num 不正常时结束循环：

```
#include <math.h>      // 为了使用 isnormal()
...
float num = 1.7e-19;
float numprev = num;

while (isnormal(num)) // 当 num 为全精度的 float 类型值
{
    numprev = num;
    num /= 13.7f;
}
```

简而言之，数学库为更好地控制如何计算浮点数，提供了扩展支持。

B.8.5　对复数的支持

复数是有实部和虚部的数。实部是普通的实数，如浮点类型表示的数。虚部表示一个虚数。虚数是-1 的平方根的倍数。在数学中，复数通常写作类似 4.2 + 2.0i 的形式，其中 i 表示-1 的平方根。

C99 支持 3 种复数类型（在 C11 中为可选）：

■ float _Complex

- double _Complex
- long double _Compplex

例如，储存 float _Complex 类型的值时，使用与两个 float 类型元素的数组相同的内存布局，实部值储存在第 1 个元素中，虚部值储存在第 2 个元素中。

C99 和 C11 还支持下面 3 种虚类型：

- float _Imaginary
- double _Imaginary
- long double _Imaginary

包含了 complex.h 头文件，就可以用 complex 代替_Complex，用 imaginary 代替_Imaginary。

为复数类型定义的算术运算遵循一般的数学规则。例如，(a+b*I)*(c+d*I)即是(a*c-b*d)+(b*c+a*d)*I。

complex.h 头文件定义了一些宏和接受复数参数并返回复数的函数。特别是，宏 I 表示-1 的平方根。因此，可以编写这样的代码：

```
double complex c1 = 4.2 + 2.0 * I;
float imaginary c2= -3.0 * I;
```

C11 提供了另一种方法，通过 CMPLX()宏给复数赋值。例如，如果 re 和 im 都是 double 类型的值，可以这样做：

```
double complex c3 = CMPLX(re, im);
```

这种方法的目的是，宏在处理不常见的情况（如，im 是无穷大或非数）时比直接赋值好。

complex.h 头文件提供了一些复数函数的原型，其中许多复数函数都有对应 math.h 中的函数，其函数名即是对应函数名前加上 c 前缀。例如，csin()返回其复数参数的复正弦。其他函数与特定的复数特性相关。例如，creal()函数返回一个复数的实部，cimag()函数返回一个复数的虚部。也就是说，给定一个 double conplex 类型的 z，下面的代码为真：

```
z = creal(z) + cimag(z) * I;
```

如果熟悉复数，需要使用复数，请详细阅读 complex.h 中的内容。

下面的示例演示了对复数的一些支持：

```
// complex.c -- 复数
#include <stdio.h>
#include <complex.h>
void show_cmlx(complex double cv);
int main(void)
{
    complex double v1 = 4.0 + 3.0*I;
    double re, im;
    complex double v2;
    complex double sum, prod, conjug;

    printf("Enter the real part of a complex number: ");
    scanf("%lf", &re);
    printf("Enter the imaginary part of a complex number: ");
    scanf("%lf", &im);
    // CMPLX()是 C11 中的一个特性
    // v2 = CMPLX(re, im);
    v2 = re + im * I;
    printf("v1: ");
```

```
        show_cmlx(v1);
        putchar('\n');
        printf("v2: ");
        show_cmlx(v2);
        putchar('\n');
        sum = v1 + v2;
        prod = v1 * v2;
        conjug =conj(v1);
        printf("sum: ");
        show_cmlx(sum);
        putchar('\n');
        printf("product: ");
        show_cmlx(prod);
        putchar('\n');
        printf("complex congjugate of v1: ");
        show_cmlx(conjug);
        putchar('\n');

        return 0;
    }
    void show_cmlx(complex double cv)
    {
        printf("(%.2f, %.2fi)", creal(cv), cimag(cv));
        return;
    }
```

如果使用 C++,会发现 C++的 complex 头文件提供一种基于类的方式处理复数,这与 C 的 complex.h 头文件使用的方法不同。

B.9　参考资料 IX：C 和 C++的区别

在很大程度上，C++是 C 的超集，这意味着一个有效的 C 程序也是一个有效的 C++程序。C 和 C++的主要区别是，C++支持许多附加特性。但是，C++中有许多规则与 C 稍有不同。这些不同使得 C 程序作为 C++程序编译时可能以不同的方式运行或根本不能运行。本节着重讨论这些区别。如果使用 C++的编译器编译 C 程序，就知道这些不同之处。虽然 C 和 C++的区别对本书的示例影响很小，但如果把 C 代码作为 C++程序编译的话，会导致产生错误的消息。

C99 标准的发布使得问题更加复杂，因为有些情况下使得 C 更接近 C++。例如，C99 标准允许在代码中的任意处进行声明，而且可以识别//注释指示符。在其他方面，C99 使其与 C++的差异变大。例如，新增了变长数组和关键字 restrict。C11 缩小了与 C++的差异。例如，引进了 char16_t 类型，新增了关键字_Alignas，新增了 alignas 宏与 C++的关键字匹配。C11 仍处于起步阶段，许多编译器开发商甚至都没有完全支持 C99。我们要了解 C90、C99、C11 之间的区别，还要了解 C++11 与这些标准之间的区别，以及每个标准与 C 标准之间的区别。这部分主要讨论 C99、C11 和 C++之间的区别。当然，C++也正在发展，因此，C 和 C++的异同也在不断变化。

B.9.1　函数原型

在 C++中，函数原型必不可少，但是在 C 中是可选的。这一区别在声明一个函数时让函数名后面的圆括号为空，就可以看出来。在 C 中，空圆括号说明这是前置原型，但是在 C++中则说明该函数没有参数。也就是说，在 C++中，int slice();和 int slice(void);相同。例如，下面旧风格的代码在 C 中可以接受，但是在 C++中会产生错误：

```
int slice();
int main()
{
    ...
    slice(20, 50);
    ...
}
int slice(int a, int b)
{
    ...
}
```

在 C 中，编译器假定用户使用旧风格声明函数。在 C++中，编译器假定 slice()与 slice(void)相同，且未声明 slice(int, int)函数。

另外，C++允许用户声明多个同名函数，只要它们的参数列表不同即可。

B.9.2 char 常量

C 把 char 常量视为 int 类型，而 C++将其视为 char 类型。例如，考虑下面的语句：

```
char ch = 'A';
```

在 C 中，常量'A'被储存在 int 大小的内存块中，更精确地说，字符编码被储存为一个 int 类型的值。相同的数值也储存在变量 ch 中，但是在 ch 中该值只占内存的 1 字节。

在 C++中，'A'和 ch 都占用 1 字节。它们的区别不会影响本书中的示例。但是，有些 C 程序利用 char 常量被视为 int 类型这一特性，用字符来表示整数值。例如，如果一个系统中的 int 是 4 字节，就可以这样编写 C 代码：

```
int x = 'ABCD'; /*对于int是4字节的系统，该语句出现在C程序中没问题，但是出现在C++程序中会出错 */
```

'ABCD'表示一个 4 字节的 int 类型值，其中第 1 个字节储存 A 的字符编码，第 2 个字节储存 B 的字符编码，以此类推。注意，'ABCD'和"ABCD"不同。前者只是书写 int 类型值的一种方式，而后者是一个字符串，它对应一个 5 字节内存块的地址。

考虑下面的代码：

```
int x = 'ABCD';
char c = 'ABCD';
printf("%d %d %c %c\n", x, 'ABCD', c, 'ABCD');
```

在我们的系统中，得到的输出如下：

```
1094861636 1094861636 D D
```

该例说明，如果把'ABCD'视为 int 类型，它是一个 4 字节的整数值。但是，如果将其视为 char 类型，程序只使用最后一个字节。在我们的系统中，尝试用%s 转换说明打印'ABCD'会导致程序奔溃，因为'ABCD'的数值（1094861636）已超出该类型可表示的范围。

可以这样使用的原因是 C 提供了一种方法可单独设置 int 类型中的每个字节，因为每个字符都对应一个字节。但是，由于要依赖特定的字符编码，所以更好的方法是使用十六进制的整型常量，因为每两位十六进制数对应一个字节。第 15 章详细介绍过相关内容（C 的早期版本不提供十六进制记法，这也许是多字符常量技术首先得到发展的原因）。

B.9.3 const 限定符

在 C 中，全局的 const 具有外部链接，但是在 C++中，具有内部链接。也就是说，下面 C++的声明：

```
const double PI = 3.14159;
```

相当于下面 C 中的声明：

```
static const double PI = 3.14159;
```

假设这两条声明都在所有函数的外部。C++规则的意图是为了在头文件更加方便地使用 const。如果 const 变量是内部链接，每个包含该头文件的文件都会获得一份 const 变量的备份。如果 const 变量是外部链接，就必须在一个文件中进行定义式声明，然后在其他文件中使用关键字 extern 进行引用式声明。

顺带一提，C++可以使用关键字 extern 使一个 const 值具有外部链接。所以两种语言都可以创建内部链接和外部链接的 const 变量。它们的区别在于默认使用哪种链接。

另外，在 C++中，可以用 const 来声明普通数组的大小：

```
const int ARSIZE = 100;
double loons[ARSIZE]; /* 在 C++中，与 double loons[100];相同 */
```

当然，也可以在 C99 中使用相同的声明，不过这样的声明会创建一个变长数组。

在 C++中，可以使用 const 值来初始化其他 const 变量，但是在 C 中不能这样做：

```
const double RATE = 0.06;          // C++和 C 都可以
const double STEP = 24.5;          // C++和 C 都可以
const double LEVEL = RATE * STEP;  // C++可以，C 不可以
```

B.9.4　结构和联合

声明一个有标记的结构或联合后，就可以在 C++中使用这个标记作为类型名：

```
struct duo
{
    int a;
    int b;
};
struct duo m; /* C 和 C++都可以 */
duo n; /* C 不可以，C++可以*/
```

结果是结构名会与变量名冲突。例如，下面的程序可作为 C 程序编译，但是作为 C++程序编译时会失败。因为 C++把 printf()语句中的 duo 解释成结构类型而不是外部变量：

```
#include <stdio.h>
float duo = 100.3;
int main(void)
{
    struct duo { int a; int b;};
    struct duo y = { 2, 4};
    printf ("%f\n", duo); /* 在 C 中没问题，但是在 C++不行 */
    return 0;
}
```

在 C 和 C++中，都可以在一个结构的内部声明另一个结构：

```
struct box
{
    struct point {int x; int y; } upperleft;
    struct point lowerright;
};
```

在 C 中，随后可以使用任意使用这些结构，但是在 C++中使用嵌套结构时要使用一个特殊的符号：

```
struct box ad;         /* C 和 C++都可以 */
```

```
struct point dot;      /* C 可以，C++不行 */
box::point dot;        /* C 不行，C++可以 */
```

B.9.5 枚举

C++使用枚举比 C 严格。特别是，只能把 enum 常量赋给 enum 变量，然后把变量与其他值作比较。不经过显式强制类型转换，不能把 int 类型值赋给 enum 变量，而且也不能递增一个 enum 变量。下面的代码说明了这些问题：

```
enum sample {sage, thyme, salt, pepper};
enum sample season;
season = sage;              /* C 和 C++都可以 */
season = 2;                 /* 在 C 中会发出警告，在 C++中是一个错误 */
season = (enum sample) 3;   /* C 和 C++都可以*/
season++;              /* C 可以，在 C++中是一个错误 */
```

另外，在 C++中，不使用关键字 enum 也可以声明枚举变量：

```
enum sample {sage, thyme, salt, pepper};
sample season;     /* C++可以，在 C 中不可以 */
```

与结构和联合的情况类似，如果一个变量和 enum 类型的同名会导致名称冲突。

B.9.6 指向 void 的指针

C++可以把任意类型的指针赋给指向 void 的指针，这点与 C 相同。但是不同的是，只有使用显式强制类型转换才能把指向 void 的指针赋给其他类型的指针。下面的代码说明了这一点：

```
int ar[5] = {4, 5, 6,7, 8};
int * pi;
void * pv;
pv = ar;           /* C 和 C++都可以 */
pi = pv;           /* C 可以，C++不可以 */
pi = (int * ) pv;  /* C 和 C++都可以 */
```

C++与 C 的另一个区别是，C++可以把派生类对象的地址赋给基类指针，但是在 C 中没有这里涉及的特性。

B.9.7 布尔类型

在 C++中，布尔类型是 bool，而且 ture 和 false 都是关键字。在 C 中，布尔类型是_Bool，但是要包含 stdbool.h 头文件才可以使用 bool、true 和 false。

B.9.8 可选拼写

在 C++中，可以用 or 来代替||，还有一些其他的可选拼写，它们都是关键字。在 C99 和 C11 中，这些可选拼写都被定义为宏，要包含 iso646.h 才能使用它们。

B.9.9 宽字符支持

在 C++中，wchar_t 是内置类型，而且 wchar_t 是关键字。在 C99 和 C11 中，wchar_t 类型被定义在多个头文件中（stddef.h、stdlib.h、wchar.h、wctype.h）。与此类似，char16_t 和 char32_t 都是 C++11 的关键字，但是在 C11 中它们都定义在 uchar.h 头文件中。

C++通过 iostream 头文件提供宽字符 I/O 支持（wchar_t、char16_t 和 char32_t），而 C99 通过 wchar.h 头文件提供一种完全不同的 I/O 支持包。

B.9.10　复数类型

C++在 complex 头文件中提供一个复数类来支持复数类型。C 有内置的复数类型，并通过 complex.h 头文件来支持。这两种方法区别很大，不兼容。C 更关心数值计算社区提出的需求。

B.9.11　内联函数

C99 支持了 C++的内联函数特性。但是，C99 的实现更加灵活。在 C++中，内联函数默认是内部链接。在 C++中，如果一个内联函数多次出现在多个文件中，该函数的定义必须相同，而且要使用相同的语言记号。例如，不允许在一个文件的定义中使用 int 类型形参，而在另一个文件的定义中使用 int32_t 类型形参。即使用 typedef 把 int32_t 定义为 int 也不能这样做。但是在 C 中可以这样做。另外，在第 15 章中介绍过，C 允许混合使用内联定义和外部定义，而 C++不允许。

B.9.12　C++11 中没有的 C99/C11 特性

虽然在过去 C 或多或少可以看作是 C++的子集，但是 C99 标准增加了一些 C++没有的新特性。下面列出了一些只有 C99/C11 中才有的特性：

- 指定初始化器；
- 复合初始化器（Compound initializer）；
- 受限指针（*Restricted pointer*）（即 restric 指针）；
- 变长数组；
- 伸缩型数组成员；
- 带可变数量参数的宏。

> **注意**
>
> 以上所列只是在特定时期内的情况，随着时间的推移和 C、C++的不断发展，列表中的项会有所增减。例如，C++14 新增的一个特性就与 C99 的变长数组类似。

异步社区
人 民 邮 电 出 版 社
www.epubit.com.cn

欢迎来到异步社区！

异步社区的来历

异步社区（www.epubit.com.cn）是人民邮电出版社旗下 IT 专业图书旗舰社区，于 2015 年 8 月上线运营。

异步社区依托于人民邮电出版社 20 余年的 IT 专业优质出版资源和编辑策划团队，打造传统出版与电子出版和自出版结合、纸质书与电子书结合、传统印刷与 POD 按需印刷结合的出版平台，提供最新技术资讯，为作者和读者打造交流互动的平台。



社区里都有什么？

购买图书

我们出版的图书涵盖主流 IT 技术，在编程语言、Web 技术、数据科学等领域有众多经典畅销图书。社区现已上线图书 1000 余种，电子书 400 多种，部分新书实现纸书、电子书同步出版。我们还会定期发布新书书讯。

下载资源

社区内提供随书附赠的资源，如书中的案例或程序源代码。

另外，社区还提供了大量的免费电子书，只要注册成为社区用户就可以免费下载。

与作译者互动

很多图书的作译者已经入驻社区，您可以关注他们，咨询技术问题；可以阅读不断更新的技术文章，听作译者和编辑畅聊好书背后有趣的故事；还可以参与社区的作者访谈栏目，向您关注的作者提出采访题目。

灵活优惠的购书

您可以方便地下单购买纸质图书或电子图书，纸质图书直接从人民邮电出版社书库发货，电子书提供多种阅读格式。

对于重磅新书，社区提供预售和新书首发服务，用户可以第一时间买到心仪的新书。

用户帐户中的积分可以用于购书优惠。100 积分 =1 元，购买图书时，在 0 使用积分 里填入可使用的积分数值，即可扣减相应金额。

特 别 优 惠

购买本书的读者专享**异步社区购书优惠券**。

使用方法：注册成为社区用户，在下单购书时输入 57AWG 使用优惠码，然后点击"使用优惠码"，即可享受电子书8折优惠（本优惠券只可使用一次）。

纸电图书组合购买

社区独家提供纸质图书和电子书组合购买方式，价格优惠，一次购买，多种阅读选择。

软技能：代码之外的生存指南

纸质版 ￥59.00 ￥46.02（7.8折）

电子版 ￥35.00

电子版 + 纸质版 ￥59.00

现在购买　下载PDF样章

社区里还可以做什么？

提交勘误

您可以在图书页面下方提交勘误，每条勘误被确认后可以获得100积分。热心勘误的读者还有机会参与书稿的审校和翻译工作。

写作

社区提供基于 Markdown 的写作环境，喜欢写作的您可以在此一试身手，在社区里分享您的技术心得和读书体会，更可以体验自出版的乐趣，轻松实现出版的梦想。

如果成为社区认证作译者，还可以享受异步社区提供的作者专享特色服务。

会议活动早知道

您可以掌握 IT 圈的技术会议资讯，更有机会免费获赠大会门票。

加入异步

扫描任意二维码都能找到我们：



异步社区



微信服务号



微信订阅号



官方微博



QQ 群：368449889

社区网址：www.epubit.com.cn

官方微信：异步社区

官方微博：@ 人邮异步社区，@ 人民邮电出版社 - 信息技术分社

投稿 & 咨询：contact@epubit.com.cn

C Primer Plus

（第6版）中文版



本书是一本经过仔细测试、精心设计的完整C语言教程，它涵盖了C语言编程中的核心内容。本书作为计算机科学的经典著作，讲解了包含结构化代码和自顶向下设计在内的程序设计原则。

与以前的版本一样，作者的目标仍旧是为读者提供一本入门型、条理清晰、见解深刻的C语言教程。作者把基础的编程概念与C语言的细节很好地融合在一起，并通过大量短小精悍的示例同时演示一两个概念，通过学以致用的方式鼓励读者掌握新的主题。

每章末尾的复习题和编程练习题进一步强化了最重要的信息，有助于读者消化那些难以理解的概念。本书采用了友好、易于使用的编排方式，不仅适合打算认真学习C语言编程的学生阅读，也适合那些精通其他编程语言，但希望更好地掌握C语言这门核心语言的开发人员阅读。

本书在之前版本的基础之上进行了全新升级，它涵盖了C语言最新的进展以及C11标准的详细内容。本书还提供了大量深度与广度齐备的教学技术和工具，来提高你的学习。

- 详细完整地讨论了C语言的基础特性和附加特性；
- 清晰解释了使用C语言不同部分的时机，以及原因；
- 通过简洁、简单的示例加强读者的动手练习，以帮助一次理解一两个概念；
- 囊括了数百个实用的代码示例；
- 每章末尾的复习题和编程练习可以检测你的理解情况；
- 涵盖了C泛型编程，以提供最大的灵活性。



读者可通过http://www.epubit.com.cn/book/details/1848下载该书的源代码。

异步社区 www.epubit.com.cn
新浪微博 @人邮异步社区
投稿/反馈邮箱 contact@epubit.com.cn

封面设计：董志桢


ISBN 978-7-115-39059-2

分类建议：计算机／程序设计／C
人民邮电出版社网址：www.ptpress.com.cn